版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Broadview
www.broadview.com.cn

大数据丛书

大数据

处理之道

何金池　编著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

大数据丛书

大数据处理之道

何金池 编著

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

内容简介

本书覆盖了当前大数据处理领域的热门技术，包括Hadoop、Spark、Storm、Dremel、Drill等，详细分析了各种技术的应用场景和优缺点；同时阐述了大数据下的日志分析系统，重点讲解了ELK日志处理方案；最后分析了大数据处理技术的发展趋势。

本书采用幽默的表述风格，使读者容易理解、轻松掌握；重点从各种技术的起源、设计思想、架构等方面阐述，以帮助读者从根源上悟出大数据处理之道。

本书适合大数据开发、大数据测试人员，以及其他软件开发或者管理人员和计算爱好者阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

大数据处理之道 / 何金池编著. —北京：电子工业出版社，2016.10
（大数据丛书）

ISBN 978-7-121-28723-7

Ⅰ. ①大… Ⅱ. ①何… Ⅲ. ①数据处理 Ⅳ. ① TP274

中国版本图书馆CIP数据核字（2016）第204120号

责任编辑：刘　皎
特约编辑：赵树刚
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：720×1000　1/16　　印张：17.75　　字数：341千字
版　　次：2016年10月第1版
印　　次：2016年10月第1次印刷
定　　价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

前言

近年来，“大数据”已然成为IT界如火如荼的词，与“云计算”并驾齐驱，成为带动IT行业发展的两列高速火车。尤其是在物联网快速发展的时代，数据已经被称为新的资源，是支撑物联网发展的基石。

那么，如何把“死”的数据变成真正有效的“资源”，成为近年来IT界人士共同思考的问题。一时间，各种大数据处理技术如井喷一般涌现。Hadoop、Spark、Storm、Dremel、Drill等大数据解决方案争先恐后地展现出来。需要说明的是，这里所有的方案并不是一种技术，而是数种甚至数十种技术的组合。就拿Hadoop来说，Hadoop只是“领头羊”，关键成员还有MapReduce、HDFS、Hive、HBase、Pig、ZooKeeper等，大有“八仙过海，各显神通”的气势和场面。

本书首先横向总结性地阐述了各种大数据处理技术，重点从缘起缘落、设计思想、架构原理等角度剖析了各种技术，分析了各种技术的优缺点和适用场景。本书并不涉及软件的安装等，因为如何安装和使用，在网络上搜索即可，着实没有必要浪费读者的时间和金钱。在这一部分，第1篇为Hadoop军营；第2篇为Spark星火燎原；第3篇讲述了其他大数据处理技术，如Storm、Dremel、Drill等。

其次阐述了大数据下的日志分析技术。在大数据时代，日志分析方案呈现出遍地开花的景象。如果将大数据处理系统比作一个可能得病的人，那么日志分析就是负责看病的医生，要想让大数据处理系统健康、平稳地运行，日志分析和监控非常重要。这一部分重点阐述了日志分析技术中如日中天的方案ELK。

最后展望了大数据处理技术的发展趋势。大数据处理技术发展迅猛，数据量越来越大，技术的革新在所难免。

作为大数据研发人员，只有时刻学习新技术，方能立于技术前沿。

由于时间仓促，书中难免出现不足之处，恳请读者指正。本书编写过程中得到了团队其他成员的支持，贡献力量的有张帅、王占伟、李峰、欧立奇等，在此衷心感谢朋友和家人的鼎力支持。

闲言少叙，直接上干货吧！亲爱的读者朋友，请吧……

目录

0 “疯狂”的大数据 1

0.1 大数据时代 1

0.2 数据就是“金库” 3

0.3 让大数据“活”起来 4

第 1 篇 Hadoop 军营

1 Hadoop 一石激起千层浪 7

1.1 Hadoop 诞生——不仅仅是玩具 7

1.2 Hadoop 发展——各路英雄集结 8

1.3 Hadoop 和它的小伙伴们 10

1.4 Hadoop 应用场景 12

1.5 小结 13

2 MapReduce 奠定基石 14

2.1 MapReduce 设计思想 14

2.2 MapReduce 运行机制 19

2.2.1 MapReduce 的组成 …… 19
2.2.2 MapReduce 作业运行流程 …… 20
2.2.3 JobTracker 解剖 …… 26
2.2.4 TaskTracker 解剖 …… 34
2.2.5 失败场景分析 …… 42
2.3 MapReduce 实例分析 …… 43
2.3.1 运行 WordCount 程序 …… 44
2.3.2 WordCount 源码分析 …… 45
2.4 小结 …… 48
3 分布式文件系统 49
3.1 群雄并起的 DFS …… 49
3.2 HDFS 文件系统 …… 51
3.2.1 HDFS 设计与架构 …… 52
3.2.2 HDFS 操作与 API …… 56
3.2.3 HDFS 的优点及适用场景 …… 60
3.2.4 HDFS 的缺点及改进策略 …… 61
3.3 小结 …… 62
4 Hadoop 体系的“四剑客” 63
4.1 数据仓库工具 Hive …… 63
4.1.1 Hive 缘起何处 …… 63
4.1.2 Hive 和数据库的区别 …… 65
4.1.3 Hive 设计思想与架构 …… 66
4.1.4 适用场景 …… 74
4.2 大数据仓库 HBase …… 74

4.2.1 HBase 因何而生 74
4.2.2 HBase 的设计思想和架构 77
4.2.3 HBase 优化技巧 84
4.2.4 HBase 和 Hive 的区别 86
4.3 Pig 编程语言 87
4.3.1 Pig 的缘由 87
4.3.2 Pig 的基本架构 88
4.3.3 Pig 与 Hive 的对比 90
4.3.4 Pig 的执行模式 90
4.3.5 Pig Latin 语言及其应用 91
4.4 协管员 ZooKeeper 96
4.4.1 ZooKeeper 是什么 96
4.4.2 ZooKeeper 的作用 97
4.4.3 ZooKeeper 的架构 98
4.4.4 ZooKeeper 的数据模型 100
4.4.5 ZooKeeper 的常用接口及操作 102
4.4.6 ZooKeeper 的应用场景分析 105
4.5 小结 108

5 Hadoop 资源管理与调度 110

5.1 Hadoop 调度机制 110
5.1.1 FIFO 111
5.1.2 计算能力调度器 111
5.1.3 公平调度器 113
5.2 Hadoop YARN 资源调度 114

5.2.1 YARN 产生的背景 114
5.2.2 Hadoop YARN 的架构 116
5.2.3 YARN 的运作流程 118
5.3 Apache Mesos 资源调度 120
5.3.1 Apache Mesos 的起因 120
5.3.2 Apache Mesos 的架构 121
5.3.3 基于 Mesos 的 Hadoop 123
5.4 Mesos 与 YARN 对比 127
5.5 小结 128

6 Hadoop 集群管理之道 129

6.1 Hadoop 集群管理与维护 129
6.1.1 Hadoop 集群管理 129
6.1.2 Hadoop 集群维护 131
6.2 Hadoop 集群调优 132
6.2.1 Linux 文件系统调优 132
6.2.2 Hadoop 通用参数调整 133
6.2.3 HDFS 相关配置 133
6.2.4 MapReduce 相关配置 134
6.2.5 Map 任务相关配置 136
6.2.6 HBase 搭建重要的 HDFS 参数 137
6.3 Hadoop 集群监控 137
6.3.1 Apache Ambari 监控 137
6.3.2 Ganglia 监控 Hadoop 138
6.4 小结 138

第 2 篇 Spark 星火燎原

7 Spark 宝刀出鞘 141

7.1 Spark 的历史渊源 141

7.1.1 Spark 的诞生 141

7.1.2 Spark 的发展 142

7.2 Spark 和 Hadoop MapReduce 对比 143

7.3 Spark 的适用场景 145

7.4 Spark 的硬件配置 146

7.5 Spark 架构 147

7.5.1 Spark 生态架构 147

7.5.2 Spark 运行架构 149

7.6 小结 151

8 Spark 核心 RDD 153

8.1 RDD 简介 153

8.1.1 什么是 RDD 153

8.1.2 为什么需要 RDD 154

8.1.3 RDD 本体的设计 154

8.1.4 RDD 与分布式共享内存 155

8.2 RDD 的存储级别 155

8.3 RDD 依赖与容错 157

8.3.1 RDD 依赖关系 157

8.3.2 RDD 容错机制 160

8.4 RDD 操作与接口 161

8.4.1 RDD Transformation 操作与接口 162
8.4.2 RDD Action 操作与接口 164
8.5 RDD 编程示例 165
8.6 小结 166
9 Spark 运行模式和流程 167
9.1 Spark 运行模式 167
9.1.1 Spark 的运行模式列表 167
9.1.2 Local 模式 168
9.1.3 Standalone 模式 169
9.1.4 Spark on Mesos 模式 171
9.1.5 Spark on YARN 模式 173
9.1.6 Spark on EGO 模式 175
9.2 Spark 作业流程 177
9.2.1 YARN-Client 模式的作业流程 178
9.2.2 YARN-Cluster 模式的作业流程 179
9.3 小结 181
10 Shark 和 Spark SQL 183
10.1 从 Shark 到 Spark SQL 183
10.1.1 Shark 的撤退是进攻 183
10.1.2 Spark SQL 接力 185
10.1.3 Spark SQL 与普通 SQL 的区别 186
10.2 Spark SQL 应用架构 187
10.3 Spark SQL 之 DataFrame 188
10.3.1 什么是 DataFrame 188

10.3.2 DataFrame 的创建 188
10.3.3 DataFrame 的使用 190
10.4 Spark SQL 运行过程分析 190
10.5 小结 192

11 Spark Streaming 流数据处理新贵 193

11.1 Spark Streaming 是什么 193
11.2 Spark Streaming 的架构 194
11.3 Spark Streaming 的操作 195
11.3.1 Spark Streaming 的 Transformation 操作 196
11.3.2 Spark Streaming 的 Window 操作 197
11.3.3 Spark Streaming 的 Output 操作 198
11.4 Spark Streaming 性能调优 198
11.5 小结 200

12 Spark GraphX 图计算系统 201

12.1 图计算系统 201
12.1.1 图存储模式 202
12.1.2 图计算模式 203
12.2 Spark GraphX 的框架 206
12.3 Spark GraphX 的存储模式 207
12.4 Spark GraphX 的图运算符 208
12.5 小结 211

13 Spark Cluster 管理 212

13.1 Spark Cluster 部署 212
13.2 Spark Cluster 管理与监控 213

13.2.1 内存优化机制 213
13.2.2 Spark 日志系统 213
13.3 Spark 高可用性 215
13.4 小结 216

第 3 篇 其他大数据处理技术

14 专为流数据而生的 Storm 218
14.1 Storm 起因 218
14.2 Storm 的架构与组件 220
14.3 Storm 的设计思想 222
14.4 Storm 与 Spark 的区别 224
14.5 Storm 的适用场景 225
14.6 Storm 的应用 226
14.7 小结 227
15 Dremel 和 Drill 228
15.1 Dremel 和 Drill 的历史背景 228
15.2 Dremel 的原理与应用 230
15.3 Drill 的架构与流程 232
15.4 Dremel 和 Drill 的适用场景与应用 234
15.5 小结 234

第 4 篇 大数据下的日志分析系统

16 日志分析解决方案 236
16.1 百花齐放的日志处理技术 236

16.2 日志处理方案 ELK 238
16.2.1 ELK 的三大金刚 238
16.2.2 ELK 的架构 240
16.2.3 ELK 的组网形式 242
16.3 Logstash 日志收集解析 245
16.3.1 Input Plugins 及应用示例 246
16.3.2 Filter Plugins 及应用示例 248
16.3.3 Output Plugins 及应用示例 249
16.4 ElasticSearch 存储与搜索 250
16.4.1 ElasticSearch 的主要概念 251
16.4.2 ElasticSearch Rest API 252
16.5 Kibana 展示 253
16.6 小结 255

17 ELK 集群部署与应用 256

17.1 ELK 集群部署与优化 256
17.1.1 ELK HA 集群部署 256
17.1.2 ElasticSearch 优化 257
17.2 如何开发自己的插件 259
17.3 ELK 在大数据运维系统中的应用 261
17.4 ELK 实战应用 262
17.4.1 ELK 监控 Spark 集群 262
17.4.2 ELK 监控系统资源状态 263
17.4.3 ELK 辅助日志管理和故障排查 263
17.5 小结 264

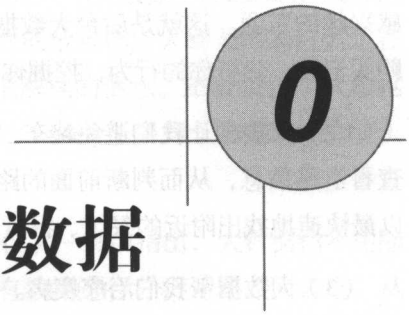
第 5 篇　数据分析技术前景展望

18　大数据处理的思考与展望　266

18.1　大数据时代的思考……266

18.2　大数据处理技术的发展趋势……267

18.3　小结……270



“疯狂”的大数据

0.1 大数据时代

大数据？多么时髦的词！就连农民在看新闻联播的时候，都经常听到“大数据告诉我们……”。只要能涉及大数据、云计算的，各个行业都不会放过，比如汽车、在线教育、饮食、传媒、医疗、金融、零售业、农业等。是的，“大数据”已经不是IT界的专有名词了，而是与我们的生活息息相关的事物，我们已经大踏步地迈进了大数据时代。相信对于大多数人来说，第一反应就是：大数据最多算是无形的资产，要说具有划时代意义，还是有点夸张了吧。笔者个人认为一点也不夸张。要特别说明的是，这种观点不是笔者杜撰的，而是很多专家学者不约而同提出来的，最具典型性的是有哈佛大学、牛津大学、耶鲁大学和新加坡国立大学等多个互联网研究重镇任教经历的维克托·迈尔·舍恩伯格教授。他在2012年出版了一部著作《大数据时代》(*Big Data:A Revolution That Will Transform How We Live, Work, and Think*)，引起了很大的反响。此书观点既大胆又新颖，他提到“大数据时代，我们关注的不是随机样本，而是全体数据；不是精确性，而是混杂性；不是因果关系，而是相关关系”。我们已经身处大数据时代。

那么大数据对我们的生活造成了哪些影响？或者说带来哪些好处？让我们来简单地看几个例子。

（1）大数据帮助我们购物。相信大家都在淘宝或者京东等电商平台上有过购物的经历，如果用心的话，你就会发现，展现在你面前的都是你近期购买的或者感兴趣的东西。这就是后台大数据服务器运行的结果。它根据你的浏览记录或者购买记录，分析你的行为，挖掘你的喜好，进而为你推荐相关的产品。

（2）大数据让我们避免堵车。我们每天在公路上行驶时，可以通过智能手机查看道路信息，从而判断前面的路是否畅通，进而选择最佳的行车路线。同时可以最快速地找出附近的餐馆、银行、加油站等信息。

（3）大数据帮我们治疗疾病。很多先进的医院已经引进机器人来为患者诊疗，甚至做手术。大数据告诉我们各种症状的真正原因和解决方案。相信大家都读过一则新闻，科技巨头 IBM 跨界医疗对抗癌症，通过 IBM Watson 计算系统，几分钟就可以处理完一支专家团队通常需要几小时或几天来处理的数据。其实，目前已经有医院试点用机器人来做手术和制订治疗方案，虽然没有得到大规模的应用，但大数据医疗已经成为一种趋势。

（4）大数据帮助我们娱乐。典型的例子是 Google AlphaGo 和韩国围棋大师李世石对弈，最终以 4:1 获胜。Google AlphaGo 通过收集李世石之前所有的棋局，加上专家的修缮，最重要的是结合机器深度学习的大数据处理算法，最终获胜。

（5）大数据帮助反腐。是的，大数据可以对一个人的财产情况进行分析，即时显示其清廉与否。还可以迅速查出偷税漏税等情况（已有例证）。

（6）大数据可以帮助警察破案。这个貌似不太稀奇，大数据几秒钟内就可以找到犯罪嫌疑人的车辆，甚至其最近出现的地点。满大街都是“天眼”（摄像头），大数据让这个世界真正做到了“天网恢恢，疏而不漏”。

（7）大数据帮助我们做健康检查。健身腕带可以收集有关我们走路或者慢跑的数据，例如我们走了多少步、每天燃烧了多少卡路里、我们的睡眠模式或者其他数据，然后结合这些数据与健康记录来改善我们的健康状况。国外有很多私人健康医生推荐客户使用这样的“健康手环”，以此来记录客户的身体情况。

类似的案例不胜枚举，相信读者也感觉到大数据正在无时无刻地影响着我们的生活。正如马云所说的，“很多人还没搞清楚什么是 PC 互联网，移动互联网来了；我们还没搞清楚移动互联网的时候，大数据时代又来了”。毫不夸张地说，我们生活在一个疯狂的大数据时代，大数据会让我们的生活越来越舒适、便利。大数据

作为新一轮信息战的主角，将创造无限商机，既便利，又危及每个人的生活。

那么，什么是大数据呢？我们来看看官方的定义：大数据（Bigdata），或称巨量数据、海量数据、大资料，指的是所涉及的数据量规模巨大到无法通过人工在合理时间内截取、管理、处理并整理成人类所能解读的信息。IBM 提出了大数据的 5V 特点：Volume（大量）、Velocity（高速）、Variety（多样）、Value（价值）、Veracity（真实性）。

维克托·迈尔·舍恩伯格教授在《大数据时代》一书中指出，大数据指不用随机分析法（抽样调查）这样的捷径，而采用所有数据进行分析处理。简言之，从各种类型的数据中快速获得有价值信息的能力就是大数据技术。明白这一点至关重要，也正是这一点促使该技术具备了走向实际应用的潜力。

0.2 数据就是“金库”

从大数据的类型来看，大致可分为三类。

- 传统企业数据：包括 CRM Systems 的消费者数据、传统的 ERP 数据、库存数据及账目数据等。
- 机器和传感器数据：包括呼叫记录、智能仪表、工业设备传感器、设备日志、交易数据等。越来越多的机器配备了连续测量和报告运行情况的装置。几年前，跟踪遥测发动机运行仅限于价值数百万美元的航天飞机。现在，汽车生产商在车辆中配置了监视器，连续提供车辆机械系统整体运行情况。一旦数据可得，公司将会从中渔利。这些机器传感数据都属于大数据的范畴。
- 社交数据：包括用户行为记录、反馈数据等。如 Twitter、Facebook 这样的社交媒体平台。计算机产生的数据可能包含着关于因特网和其他使用者行动与行为的有趣信息，从而提供了对他们的愿望和需求潜在的有用认知。还有人们通过电子邮件、短信、微博等产生的文本信息。

这三种数据都价值连城，甚至有说法称，谁掌握了数据资源，谁就有了“金库”。IBM、Amazon、Google、eBay、Twitter 和 Facebook 等都试图使用这三类信息认识消费者行为，预测特定需求和整体趋势。当然，要将数据变成真正的“金库”，需要一个过程，一个分析的过程，一个总结的过程，一个预测的过程。最

终，数据能帮助人类提升预测能力。通过大数据的处理工具，提取精华，总结规律，预测行为。*Science* 杂志刊登过一篇文章，指出虽然人们的出行模式有很大不同，但我们大多数人是可以预测的。这意味着我们能够根据个体之前的行为轨迹预测其未来行踪的可能性，即 93% 的人类行为可预测。93%！可怕吗？也就是说“大数据”能预测出你 93% 会去某个地方做某一件事情。

对未来的预测功能是目前业界对大数据最看重的价值。基于之前记录下来的各种数据的深入研究，发现其中的规律特征，从而进行系统优化，甚至升级。这种基于预测的价值实现对系统（包括个人、企业机构、各种电子性工具）的长远运行来说价值重大，决定了一个系统是否具有长期的成长性和演变能力。而大数据定理告诉我们，在试验不变的条件下，重复试验多次，随机事件的频率近似于它的概率。“有规律的随机事件”在大量重复出现的条件下，往往呈现几乎必然的统计特性。举个例子，我们向上抛一枚硬币，硬币落下后哪一面朝上本来是偶然的，但当我们上抛硬币的次数足够多时，我们就会发现，硬币每一面向上的次数约占总次数的 1/2，偶然中包含着某种必然。所以有一种大胆的提法：在某些场合下，大数据让“偶然”变成“必然”，大数据让随机不再“随机”。

随着计算机处理能力的日益强大，获得的数据量越大，挖掘到的价值就越多。从目前大数据的格式和内容来看，产生的数据内容中有 90% 是非结构化数据，如短信和微博生成的信息，以及视频和音频，情况将变得越来越复杂（至今最大的数据是音频、视频和符号数据，这些数据结构松散、数量巨大，挖掘难度很大）。无论身处哪个行业，围绕大数据及管理这些信息的挑战无处不在。

据麦肯锡全球研究院测算，大数据将给美国医疗服务业带来 3000 亿美元的价值，使美国零售业净利润增长达到 60%，使制造业产品开发、组装成本下降 50%。而大数据所带来的新的信息技术应用需求将推动整个信息技术产业的创新发展。

0.3 让大数据“活”起来

如何让数据变成真正的资产，变成真正的“金库”？如何让数据“活”起来，并“火”起来？这成为人们追求和思考的话题。

法国著名雕塑家罗丹说过，“世界上并不缺少美，只是缺少发现美的眼睛”。同样，世界上并不缺少数据，而是缺少开发数据的工具。大数据不是最近才有的，

数据一直存在，只是大数据的价值近几年才被人们认识到，分析数据的能力近几年才有了革命性的突破。

有报道称，世界上每分钟产生1700TB的数据，无论是工业、金融、研究、办公、媒体还是日常生活，产生的数据均可成为大数据的一部分。但是吸引我们的不仅仅是这个庞大的数字本身，还在于我们可以用这些数据做些什么。因为数据本身不产生价值，如何分析和利用大数据对业务产生帮助才是关键。

从具体应用上来看，大数据发挥的真正价值还不是特别明显，也就是说，还没有“生产化”，企业现在热衷做的不是从大数据中提炼出价值，而更多的是存储一切能够搜集到的数据，留待以后从中挖掘价值。诚然，就目前而言，大数据处理和分析尚未成熟，尤其是在从IT手段转化为经营效益的过程中，可以说大数据很难真正找到“数据变现”的法门。

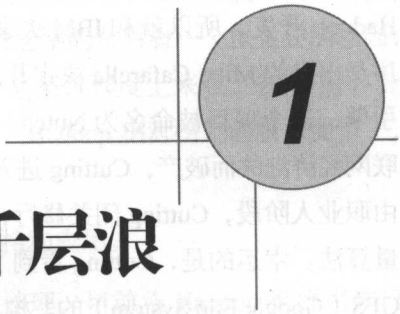
从大数据“生成”到“产生价值”，需要一个很复杂的过程：首先需要大数据采集，其次对采集之后的数据进行预处理和存储，最后对这些存储的数据进行分析和挖掘，并展现在用户面前，最终变成真正有用的资产。

目前IT界最火的两个词莫过于“大数据”和“云计算”，一时间，几乎所有的IT公司都在改革，拼命往“大数据”和“云计算”两条轨道上挤。其实大数据和云计算可以说是相辅相成的。自从有了云计算服务器，“大数据”才有了可以运行的轨道，才可以实现其真正的价值。有人将各种“大数据”的应用形象地比作一辆辆“汽车”，支撑这些“汽车”运行的“高速公路”就是云计算。最著名的实例就是Google搜索引擎。面对海量Web数据，Google于2006年首次提出“云计算”的概念。支撑Google内部各种“大数据”应用的，正是Google公司自行研发的云计算服务器。

近几年，大数据的分析技术可谓是“遍地开花”。针对不同的场景和需求，各大IT公司争先恐后地提出了各种各样的方案和技术。本书将对大数据的处理技术进行讨论，分析各项大数据处理技术的特点及适用场景，为不同的大数据场景提供切实可行的处理方案。也就是阐述怎么让大数据“活”起来。

第 1 篇

Hadoop 军营



Hadoop 一石激起千层浪

1.1 Hadoop 诞生——不仅仅是玩具

Hadoop? Hadoop 是什么？多么奇怪的名字。英语过了六级，也不曾和这个单词照过面！不乏有人在线翻译一把。其实大可不必，因为这个单词在牛津字典里面压根就不存在。它是 Hadoop 的灵魂人物 Doug Cutting 的儿子取的名字！读者可能会问，莫非是父子俩共同研发的 Hadoop？非也，非也。那时候 Cutting 的儿子仅仅三岁，这是他给自己的玩具黄色小象起的名字而已。Cutting 在创建 Hadoop 之初，正在苦思冥想为自己的产品起个什么名字的时候，突然听到他儿子叫自己的玩具“Hadoop”。Cutting 听之大喜，多么大气、响亮、顺口、上档次的名字，就它了！于是，Cutting 就在他三岁儿子未曾授权的情况下，“借用”了他儿子玩具的名字，甚至连 Hadoop 的 Logo（黄色的小象）也一起“借用”了过来。这就是 Hadoop 名字和 Logo 的来历，一个很有趣的故事。

Doug Cutting，可谓 IT 界的“弄潮儿”，人称“Hadoop 之父”，也是大数据处理的鼻祖性人物。其实，Cutting 1985 年毕业于美国斯坦福大学时，并没有打算致力于 IT 行业，因为那时候 IT 行业还处于萌芽状态，软件工程还是 Cutting 毕业之后自修的。1997 年，Cutting 因为工作需要经常查阅资料，他发现这个过程很是浪费时间和精力，因此决心改善当下的搜索技术，便每周投入两天时间，试着用 Java 把这个想法变成现实。不久之后，第一个提供全文本搜索的开源函数库诞生了，这就是 Lucene。Lucene 的伟大自不必多言，就连后来 Cutting 到 IBM 面试的

时候，IBM 还想让 Cutting 把 Lucene 继续发扬光大，可惜那时候 Cutting 一心投入 Hadoop 开发，所以就和 IBM 失之交臂（甚为遗憾）。2004 年，Cutting 和同是程序员出身的 Mike Cafarella 决定开发一款可以代替当时的主流搜索产品的开源搜索引擎，这个项目被命名为 Nutch。在此之前，Cutting 供职于 Architext，但终因互联网经济泡沫而破产，Cutting 进入了自由职业人阶段。所谓“福兮祸所伏”，在自由职业人阶段，Cutting 闭关修行，试图通过一种低开销的方式来构建网页中的大量算法。幸运的是，Cutting 看到了 2004 年 Google 发表的两篇论文，一篇是关于 GFS（Google File System）的，构思了一个可扩展的大型数据密集型应用的分布式文件系统，该文件系统可在廉价的硬件上运行，并具有可靠的容错能力，该文件系统可为用户提供极高的计算性能，而同时具备最小的硬件投资和运营成本，用于存储不同设备所产生的海量数据；另一篇是关于 MapReduce 的，它运行在 GFS 之上，是一种处理大型及超大型数据集并生成相关执行的编程模型，负责分布式大规模数据。Cutting 便利用这段“自由时间”实现了这两篇论文的想法，由此诞生了引人注目的作品——Hadoop！

随后，在持续了四年的自由职业人生涯后，Cutting 决定找一家靠谱的公司来继续完善和提升 Hadoop。他先后面试了几家 IT 届的顶尖公司，最终接受了雅虎的邀请，于 2006 年正式加入雅虎，带领 100 多人的团队完善 Hadoop 项目。在此阶段，Hadoop 的稳定性得到了极大的提升。不久之后，雅虎宣布将其旗下的搜索引擎产品部署在一个拥有 1 万个内核的 Hadoop 集群上。两年后，雅虎便基于 Hadoop 启动了第一个应用项目“webmap”（一种用来计算网页间链接关系的算法）。

这就是 Hadoop 的孕育和诞生过程，所以在笔者看来，“历史是少数人创造的，是普通大众推动的”。现在受益于 Hadoop 的人，应当牢记 Doug Cutting 这位英雄和 Google 的创新思想。

1.2 Hadoop 发展——各路英雄集结

Hadoop 诞生之后，其发展可谓一呼百应，各路 IT 界公司相继直接或间接地加入了 Hadoop 阵营，纷纷表示愿意为 Hadoop 的发展“添砖加瓦”（当然也可以从 Hadoop 带来的好处中分得一杯羹）。笔者认为造成蒸蒸日上的局面的主要原因有两点：其一是大数据处理的迫切需求，各大公司都看到了数据市场的重要性，

在 IT 日新月异的高速列车上，谁不能正确地预测发展形势，谁必将被时代所淘汰；其二要感谢雅虎大无畏的“开源精神”，当然开源之举的背后有一个很重要的原因，就是和 Google 的激烈竞争。所以，笔者认为，从某种程度上来说，在 IT 界，开源是后来者居上的最有效的办法。直到 2008 年，Hadoop 相对稳定的版本面世了，并在同年 1 月成为 Apache 顶级项目。

接下来，我们看看各路英雄对 Hadoop 的支持和应用。

百度从 2007 年开始使用 Hadoop 进行离线处理，目前将 Hadoop 集群用作日志处理。

中国移动也从 2007 年开始将 Hadoop 应用于“大云”中，规模超过 1000 台。

阿里巴巴从 2008 年开始在“云梯”中应用 Hadoop，初级规模就达到 1100 台。其用于处理电子商务相关数据，每天处理约 18 000 道作业，扫描 500TB 数据。

Cloudera 于 2009 年推出 CDH（Cloudera's Distribution including Apache Hadoop）平台，完全由开放源码软件组成。

IBM 于 2010 年提供了基于 Hadoop 的大数据分析软件——InfoSphere BigInsights，包括基础版和企业版。Platform Computing（现在为 IBM Platform Computing）于 2011 年宣布在它的 Symphony 软件中支持 Hadoop MapReduce API。

MapR Technologies 公司于 2011 年 5 月推出分布式文件系统和 MapReduce 引擎——MapR Distribution for Apache Hadoop。

EMC 于 2011 年 5 月为客户推出一种新的基于开源 Hadoop 解决方案的数据中心设备——Greenplum HD，以助其满足客户日益增长的数据分析需求并加快利用开源数据分析软件。Greenplum 是 EMC 于 2010 年 7 月收购的一家开源数据仓库公司。

Calxeda 公司（之前名为 Smooth-Stone）于 2011 年 6 月发起了“开拓者行动”，一支由 10 家软件公司组成的团队将为基于 Calxeda 推出的 ARM 系统上芯片设计的服务器提供支持，并为 Hadoop 提供低功耗服务器技术。

Cloudera 于 2011 年 8 月公布了一项有益于合作伙伴生态系统的计划——创建一个生态系统，以便硬件供应商、软件供应商及系统集成商可以一起探索如何使用 Hadoop 更好地洞察数据。

Dell 与 Cloudera 于 2011 年 8 月联合推出 Hadoop 解决方案——Cloudera Enterprise。Cloudera Enterprise 基于 Dell PowerEdge C2100 机架服务器及 Dell PowerConnect 6248 以太网交换机。

各大 IT 厂商直接或间接地接触、加入 Hadoop 的阵营，一时间，Hadoop 成为“程序猿”们茶余饭后的重要谈论话题。Hadoop 从一个开源的 Apache 基金会项目，随着越来越多的用户加入，不断地使用、贡献和完善，形成一个强大的生态系统。从 2009 年开始，随着云计算和大数据的发展，Hadoop 作为海量数据分析的最佳解决方案，开始受到众多 IT 厂商的关注，从而出现了许多 Hadoop 的商业版及支持 Hadoop 的产品，包括软件和硬件。

到现在为止，Apache Hadoop 版本分为两代，我们将第一代 Hadoop 称为 Hadoop 1.0，将第二代 Hadoop 称为 Hadoop 2.0。第一代 Hadoop 包含三个大版本，分别是 0.20.x、0.21.x 和 0.22.x。其中，0.20.x 最后演化成 1.0.x，变成了稳定版；而 0.21.x 和 0.22.x 则增加了 NameNode HA 等新的重大特性。第二代 Hadoop 包含两个版本，分别是 0.23.x 和 2.x，它们完全不同于 Hadoop 1.0，是一套全新的架构，均包含 HDFS Federation 和 YARN 两个系统。相比于 0.23.x，2.x 增加了 NameNode HA 和 Wire-compatibility 两个重大特性。

1.3 Hadoop 和它的小伙伴们

Hadoop 迅速发展壮大，由原来的 MapReduce 和 HDFS 发展到 Hadoop 生态圈，人们提到 Hadoop，更多的是代表生态圈内数十个产品，并且越来越多。那么，生态圈中有这么多小伙伴，它们的架构是怎么样的呢？接下来请 Hadoop 和它最常用（也最重要）的伙伴出场。图 1-1 为 Hadoop 生态圈组织架构示意图。

这里，我们先对 Hadoop 的常用小伙伴做一个简单的介绍，大概记住名字和主要功能即可，后续章节我们还会多次打交道，再逐个详细阐述。

MapReduce：核心成员，是一种编程模型，使得大规模数据可以并行计算。从概念上看，MapReduce 有两个部分——“Map（映射）”和“Reduce（归约）”，当然不能简单地理解为 Map 和 Reduce 两个函数。事实上，此模型包含若干个函数和接口，其主要思想最先由 Google 的工程师提出，得益于函数式编程语言的思想，以及从矢量编程语言里借来的特性。它使编程人员在不懂分布式并行编程的情况

下，也能将自己的程序运行在分布式系统上。

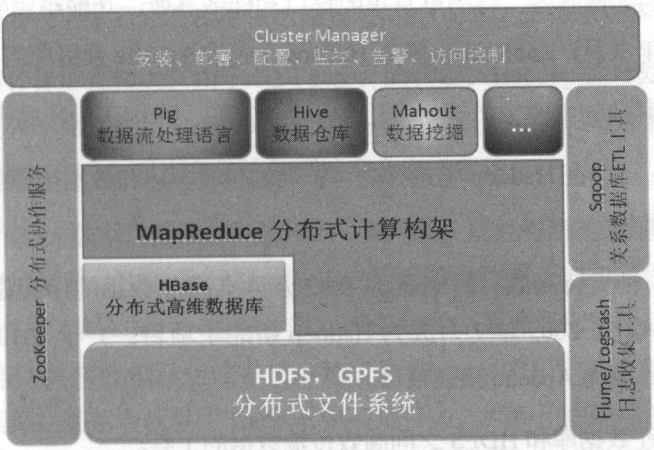


图 1-1 Hadoop 生态圈组织架构示意图

HDFS：一个分布式文件系统。分布式文件系统相信大家都不陌生，但相对于其他文件系统，HDFS 的设计有着为大数据处理“私人订制”的意思，提供高吞吐量的数据访问，非常适合大规模数据集上的应用。并且它是一个高度容错的系统，适合部署在廉价的机器上。

Hive：一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供完整的 SQL 查询功能，可以将 SQL 语句转换为 MapReduce 任务运行。其优点是学习成本低，可以通过类 SQL 语句快速实现简单的 MapReduce 统计，不必开发专门的 MapReduce 应用，非常适合数据仓库的统计分析。

HBase：一个分布式的、面向列的开源数据库，和 Google Bigtable 类似。HBase 是 Apache 的 Hadoop 项目的子项目。HBase 不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库。另一个不同点是 HBase 是基于“列”的而不是基于“行”的模式。2010 年 5 月，HBase 脱离 Hadoop 项目，成为 Apache 顶级项目。

Pig：一个高级过程语言，适合于使用 Hadoop 和 MapReduce 平台来查询大型半结构化数据集。通过允许对分布式数据集进行类似 SQL 的查询，Pig 可以简化 Hadoop 的使用。2010 年 9 月，Pig 脱离 Hadoop，成为 Apache 顶级项目。

ZooKeeper：一个分布式应用程序的协调服务，类似于 Google 的 Chubby，提

供的功能包括配置维护、名字服务、分布式同步、组服务等。其目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。2011 年 1 月，ZooKeeper 脱离 Hadoop，成为 Apache 顶级项目。

Hadoop 其他的主要成员有如下几个，限于篇幅，本书不对其展开讲解。

Mahout：一个在 Hadoop 上运行的可扩展的机器学习和数据挖掘类库（例如分类和聚类算法）。

Avro：一种支持高效、跨语言的 RPC 及永久存储数据的序列化系，主要负责数据的序列化。2009 年 7 月成为 Hadoop 新的子项目，并于 2010 年 5 月脱离 Hadoop 项目，成为 Apache 顶级项目。

Sqoop：在数据库和 HDFS 之间高效传输数据的工具。

Hadoop 家族阵营之大，不是一两本书可以介绍的，把任何一个组件拿出来单独讲解可能都需要很大篇幅，建议有兴趣的读者参照 Apache 官网介绍。

1.4 Hadoop 应用场景

任何一项技术的出现都是为了解决实际的业务问题。随着 Hadoop 生态圈的逐步扩大，其应用范围也随之扩大。就 Hadoop 技术本身来讲，适合于以下应用场景：

（1）超级大的数据。一般非实验情况下使用 Hadoop 的环境，集群规模都在上百台、甚至上千台服务器的 Cluster 中。相对而言，数据量也非常大，TB 级别的数据只能算是“起步价”，PB 级别的数据才能发挥出 Hadoop 的特长。

（2）离线数据。实时处理一直是 Hadoop 有待解决的难题。Hadoop 针对批处理作业的特点是数据吞吐量大，能够处理大规模的数据；但是带来的另一个问题就是一个 Job 启动就不能再进行更改，而且有时一个 Job 可能会运行几小时。在 MapReduce 框架下很难处理实时计算，作业都以日志分析这样的线下作业为主。另外，集群中一般都会有大量作业等待被调度，以保证资源充分利用。如果实时处理要求比较高，笔者建议部署 Storm 或者 Spark。当然，也有用户已经借用 Hadoop 的组件将 Hadoop 应用到了实时处理环境中，我们后文会提到。

（3）并行计算。在 MapReduce 框架下，非常适合处理并行计算的场景。在这

种场景下，数据量极大，一般在 PB 级别及以上，但处理数据的算法相对简单，可以采用并行的方式来提高处理速度，比如大规模的 Web 信息搜索场景。

（4）数据计算单元较大。由于 HDFS 设计的特点，Hadoop 适合处理文件块大的文件，大量的小文件使用 Hadoop 来处理效率会很低。当然，如果采用其他文件系统，比如 IBM GPFS，对小文件的处理速度会有所改善。

Hadoop 作为大数据存储及计算领域的一颗明星，已经得到越来越广泛的应用。从目前实际使用来看，Hadoop 已经成功地应用到移动数据（如中国移动、中国电信等）、电子商务（如 eBay）、能源开采、图像处理、基础架构管理、医疗保健和 IT 运维等场景。其实我们已经或多或少地成为 Hadoop 的用户或者受益者。比如某大型购物网站，其后台就采用 Hadoop 来处理日志，判断用户的习惯和行为，进而为用户推荐适合的产品。中国移动也应用 Hadoop 技术，基于 Hadoop 来部署云平台。

1.5 小结

Hadoop 自诞生以来，得到迅速的发展，并广泛应用于各个行业和领域。现在很多大型互联网公司出于成本考虑都在研究和使用 Hadoop，数据的价值正得到越来越多人的重视，而这种重视又体现出 Hadoop 存在的巨大价值。本章回顾了 Hadoop 的诞生和发展过程，对 Hadoop 的集群和架构及其生态圈进行了分析，并对 Hadoop 适合和不适合的应用场景进行了分析说明。

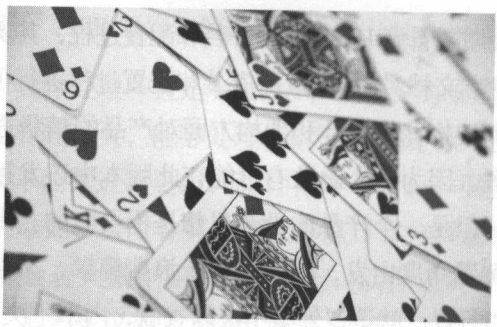
可能有的读者会期望看到 Hadoop 的安装和配置过程，其实这些资料在网上搜索即可，而且也有官方文档对 Hadoop 的安装和配置过程进行了详细说明。本书讲述的只是设计思想和架构，从设计和架构方面对产品的精华进行深入了解比学习几个函数要有用得多。因为一个函数有可能在此版本中还发挥作用，下一个版本就有可能被取消了，但是一个产品的设计思想是不会变的。因此，本书的后续章节也不会从安装和配置角度出发，因为那样既浪费篇幅，又浪费读者的时间。如果读者要学习安装和配置，就请“外事不决问谷歌，内事不决问百度”吧。

2 MapReduce 奠定基石

MapReduce 是 Hadoop 阵营中的核心，本章将对 MapReduce 进行详细的分析探讨。

2.1 MapReduce 设计思想

现在让我们来做一个游戏:假如你和朋友玩牌，但是不知道这一摞牌够不够 54 张。大家都知道，一副牌中，有 13 张红桃，13 张黑桃，13 张方块，13 张梅花，外加两张王牌。



那么你会怎么数呢？有两种方式：

第一种，你一个人从头到尾数一遍，并逐个记录下多少张红桃、多少张黑桃、多少张方块、多少张梅花，以及王牌的张数。

第二种，你将牌分发给你和你的牌友们，让他们“并行”地数，“并行”地统计，然后他们把结果反馈给你，你最后总结牌友们反馈的结果。

你会采用哪一种方法呢？第一种是比较“勤快”的人采取的“愚蠢”的做法，第二种则是“懒惰”的人采取的“聪明”的做法。你可能会说，不就 54 张牌吗，一会儿就数完了，没必要动脑筋统计汇总。是的，那假如有上千张牌、上万张牌又该如何操作呢？

近些年，人们一直争论一个富有哲学色彩的问题：是懒惰推动了科技进步，还是科技进步助长了懒惰呢？笔者认为，从某种角度来讲，科技进步弥补了人类的懒惰，所以懒惰并不见得是件坏事。有数据表明，懒惰的人一般创新意识比较强。

第一种方法是一种串行的方法，相对来说，比较浪费时间，但操作相对简单，不用沟通，不用协调，不用总结。在我们的例子中只有 54 张牌，如果有 540 张、5400 张呢？

第二种方法是一种并行的方法，相对于第一种方法来说省时，但是增加了不必要的麻烦，比如分发、沟通、反馈、总结。这其实就是 MapReduce 的思想概要。

之前说过，Hadoop 的核心思想来源于 Google 的两篇论文。Google 的那篇 MapReduce 论文里说：Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages。大致意思是，MapReduce 的灵感来源于函数式语言（比如 Lisp）中的内置函数 map 和 reduce。这句话提到了 MapReduce 思想的渊源。Lisp 是一种列表处理语言（List Processing），是一种应用于人工智能处理的符号式语言，由 MIT 的人工智能专家、图灵奖获得者 John McCarthy 于 1958 年设计发明。所以，从某种层面上讲，MapReduce 的思想早在 1958 年就被提出来了。

简单来说，在函数式语言里，Map 表示对一张列表（List）中的每个元素进行计算，Reduce 表示对一张列表中的每个元素进行迭代计算。在 MapReduce 里，Map 处理的是原始数据，自然是杂乱无章的，每条数据之间没有依赖关系，简单地说就是：你是你，我是我，你 Map 的时候不影响我，我 Map 的时候不影响你。到了 Reduce 阶段，数据是以 key 后面跟着若干个 value 来组成的，这些 value 有相关性，至少它们都在一个 key 下面，于是就符合函数式语言里 Map 和 Reduce 的基本思想了。这样我们就可以将 MapReduce 理解为：把一堆杂乱无章的数据按照

某种特征归纳起来，然后处理并得到最后的结果。Map 面对的是杂乱无章的、互不相关的数据，它解析每个数据，从中提取出 key 和 value，也就是提取数据的特征。

为了简化起见，我们可以将 MapReduce 分为两个经典的函数。

- 映射（Mapping）：对集合里的每个目标应用同一个操作。在上面的例子中，我们需要把每张牌“数”一遍，并且分别映射到红桃、黑桃、方块、梅花的张数中。经过 Mapping 之后，会输出一个 key/value 对。例如，红桃:4 张；黑桃：6 张，等等。
- 化简（Reducing）：相当于遍历集合中的元素返回一个汇总的结果。在上面的例子中，将各位牌友的反馈结果汇总，最后计算出是否是 54 张牌。注意，这里是按照相同的 key 进行汇总的，即黑桃：×× 张；红桃：×× 张，等等，黑桃、红桃等在这里充当了“key”。

具体分析，Hadoop MapReduce 用于超大规模的数据处理时，精华的思想有以下三个。

1. 并行化

三个臭皮匠赛过诸葛亮，这句古语其实就有并行计算的思想在里面，诸葛亮的计谋再厉害，也比不过三个臭皮匠并行思考。同样的道理，三台性能一般的机器，其并行计算的速度也可能超过一台强悍的服务器级别的机器的计算速度。

那么，什么样的计算任务可以进行并行化计算？如果大数据可以分为具有同样计算过程的数据块，并且这些数据块之间不存在数据依赖关系，则提高处理速度的最好办法就是并行计算。

在并行计算中，一次可执行多个指令的算法，目的是提高计算速度，以及通过扩大规模求解问题，解决大型而复杂的计算问题。其实并行计算可分为时间上的并行和空间上的并行。时间上的并行就是指流水线技术，而空间上的并行则是指用多台处理器并发地执行计算。并行计算的基本思想是用多台处理器来协同求解同一个问题，即将被求解的问题分解成若干部分，各部分均由一台独立的处理器来并行计算。并行计算系统既可以是专门设计的、含有多台处理器的超级计算机，也可以是以某种方式互连的若干台独立计算机构成的集群。通过并行计算集群完成数据的处理，再将处理的结果返回给用户。

并行计算的第一个重要问题是如何划分计算任务或者计算数据，以便对划分

的子任务或数据块同时进行计算。划分得均匀与否直接关系到大数据的处理速度。当然，结合一些均衡的调度算法，可以使得集群中的机器都能发挥最大的作用，减少闲置或者等待的时间。但在一些场合下，数据之间存在相互依赖关系，或者数据出现迭代算法，恰好无法划分。顺便提及一句，在类似这样的场合，推荐使用 Spark，因为 Spark 在具有迭代的场合性能上要比 Hadoop MapReduce 强出许多。

Map 和 Reduce 都是并行执行的，如图 2-1 所示。

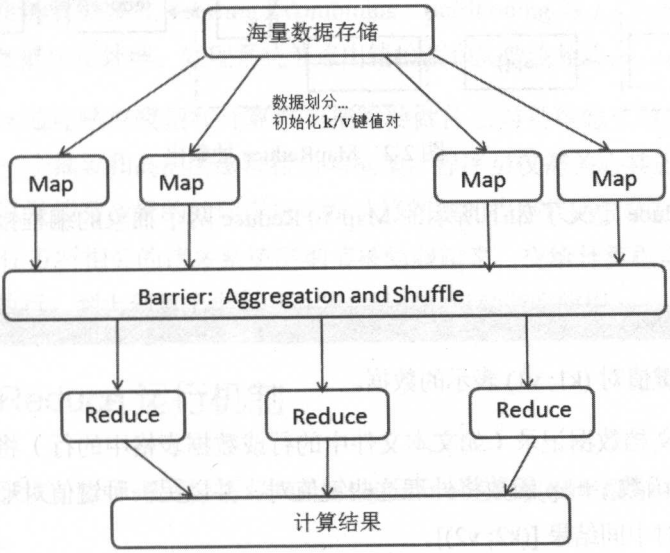


图 2-1　Map 和 Reduce 并行执行示意图

各个 map 函数对所划分的数据进行并行处理，从不同的输入数据产生不同的中间结果输出。各个 reduce 函数也各自进行并行计算，各自负责处理不同的中间结果数据集合。在进行 reduce 处理之前，必须等到所有的 map 函数执行完，因此，在进入 reduce 前需要有一个同步障（Barrier）；这个阶段也负责对 map 函数的中间结果数据进行收集整理（Aggregation & Shuffle），以便 reduce 更有效地计算最终结果。最终汇总所有 reduce 函数的输出结果即可获得最终结果。

2. 抽象化

MapReduce 将编程框架抽象化或者模型化。在整个框架中只有两个模型：Mapper 与 Reducer。在以往的并行计算方法中，缺少高层并行编程模型，在解决实际问题时，程序员的思维一直停留在方法、函数这个层面，并没有把方法和函

数抽象化。为了克服这一缺陷，MapReduce 借鉴了 Lisp 函数式语言中的思想，用 map 和 reduce 两个函数提供了高层的并行编程抽象模型，如图 2-2 所示。

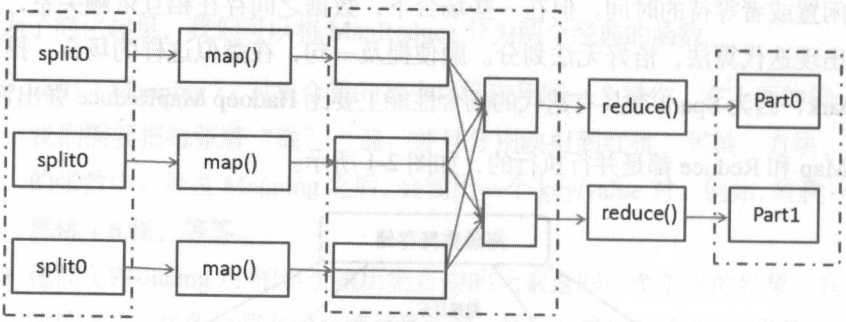


图 2-2　MapReduce 抽象化

MapReduce 定义了如下所示的 Map 和 Reduce 两个抽象的编程接口，由用户去编程实现：

```
map: (k1; v1) → [(k2; v2)]
```

输入：键值对 (k1; v1) 表示的数据。

处理：文档数据记录（如文本文件中的行或数据表格中的行）将以键值对形式传入 map 函数；map 函数将处理这些键值对，并以另一种键值对形式输出处理的一组键值对中间结果 [(k2; v2)]。

输出：键值对 [(k2; v2)] 表示的一组中间数据。

```
reduce: (k2; [v2]) -> [(k3; v3)]
```

输入：由 map 输出的一组键值对 [(k2; v2)] 将被进行合并处理，将同样主键下的不同数值合并到一张列表 [v2] 中，故 reduce 的输入为 (k2; [v2])，其中 [v2] 可以理解为一张列表。

处理：对传入的中间结果列表数据进行某种整理或进一步处理，并产生最终的某种形式的结果输出 [(k3; v3)]。

输出：最终处理结果 [(k3; v3)]。

值得一提的是，Map 和 Reduce 为程序员提供了一个清晰的操作接口抽象描述，在 MapReduce 使用过程中架构清晰明了，如果出现接口问题，也比较容易定位。

3. 架构统一，隐藏细节

MapReduce 统一架构，为程序员隐藏系统层细节。在之前的并行计算方法中，缺少统一的计算框架支持，程序员需要考虑数据存储、划分、分发、结果收集、错误恢复等诸多细节。为此，MapReduce 设计并提供了统一的计算框架，为程序员隐藏了绝大多数系统层面的处理细节。MapReduce 提供一个统一的计算框架，可完成计算任务的划分和调度、数据的分布存储和划分、处理数据与计算任务的同步、结果数据的收集整理（sorting、combining、partitioning 等）、系统通信、负载平衡、计算性能优化处理、处理系统节点出错检测和失效恢复等。

MapReduce 通过抽象模型和计算框架把需要做什么与具体怎么做区分开来，为程序员提供一个抽象和高层的编程接口与框架。程序员仅需关心其应用层的具体计算问题，以及编写少量的处理应用本身计算问题的程序代码。如何具体完成这个并行计算任务所相关的诸多系统层细节被隐藏起来，交给计算框架去处理：从分布代码的执行，到大到数千小到单个节点集群的自动调度使用。

2.2 MapReduce 运行机制

首先需要说明，以下通过分析经典的 MapReduce 运行机制来展示 MapReduce 是如何运行的。

2.2.1 MapReduce 的组成

在讲述 MapReduce 作业运行机制之前，我们先认识一下在作业运行过程中涉及的 4 个主要实体及它们的职责。

1. 客户端（Client）

提交 MapReduce 应用程序。每一个 Job 都会在客户端通过 JobClient 类将应用程序及配置参数 Configuration 打包成 JAR 包存储在 HDFS 上。

2. JobTracker

协调作业的运行。JobTracker 实际上是一个 Java 应用程序，它的主类是 JobTracker。其主要负责的事情包括：与客户端通信、接收客户端的命令（如 Submit Job、Kill Job 等）、接收 TaskTracker 心跳（分配 Task 任务队列，更新 Task 状

态，以及监测状态）、内部处理操作（对 Job 进行初始化，分解成多个 Map/Reduce Task 任务）、按照调度算法对 Job 进行排列等。

3. TaskTracker

运行作业划分后的任务。TaskTracker 也是一个 Java 应用程序，它的主类是 TaskTracker。TaskTracker 周期性地向 JobTracker 发送心跳报告，在 RPC 调用返回结果后，解析结果得到 JobTracker 下发的运行 Task 的指令，即 LaunchTaskAction，就会在 TaskTracker 节点上准备运行这个 Task。Task 的运行是在一个与 TaskTracker 进程隔离的 JVM 实例中执行的，该 JVM 实例是通过 org.apache.hadoop.mapred.Child 来创建的。所以在创建 Child VM 实例之前，需要做大量的准备工作来启动 Task 运行。

4. 文件系统

一般情况下，文件系统采用 HDFS。它是在其他实体之间共享作业和文件的文件系统，后续章节会详细分析。

这四者的互操作关系大致如图 2-3 所示。

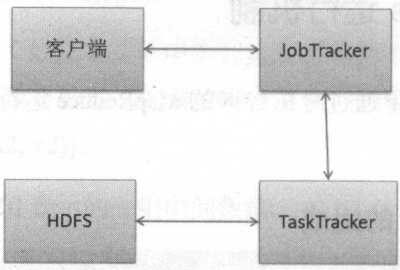


图 2-3　MapReduce 的组成部分和相互关系

其实，从 Hadoop 的总体架构上看，Hadoop MapReduce 采用 Master/Slave 结构。Master 是整个集群唯一的全局管理者，其功能包括作业管理、状态监控和任务调度等，相当于 MapReduce 中的 JobTracker。Slave 主要负责任务的执行和任务状态的报告，相当于 MapReduce 中的 TaskTracker。

2.2.2　MapReduce 作业运行流程

上面我们认识了 MapReduce 的四大实体，接下来，我们通过对整个作业流程的详细分析，来看看这四大实体是怎么相互配合的。

Hadoop 运行 MapReduce 作业的步骤主要包括：提交作业、初始化作业、分配任务、执行任务、更新进度和状态、完成作业，如图 2-4 所示。



图 2-4　MapReduce 作业运行流程

1. 提交作业

作业提交的流程和步骤如图 2-5 所示。

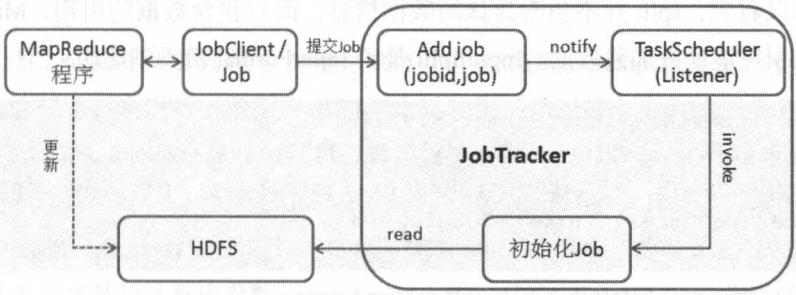


图 2-5　MapReduce 作业提交的流程和步骤

1）命令行提交

用户使用 Hadoop 命令行脚本提交 MapReduce 程序到集群，具体会调用 JobClient.runJob() 方法开始提交，最终则通过 Job 对象内部的 JobClient 对象的 submitJobInternal() 方法提交作业到 JobTracker。

2）作业上传

在提交作业到 JobTracker 之前还需要完成相关初始化工作，这些工作包括：获取用户作业的 jobId，创建 HDFS 目录，上传作业、相关依赖库，分发所需的文件到 HDFS 上，以及用户输入数据的所有分片信息等。需要注意的是，这些要上传和下载的文件都是由 DistributedCache 完成的，对用户来说是透明的，不需要用户直接干涉。

3）产生切分文件

在作业提交后，JobClient 调用 InputFormt 中的 getSplits() 方法产生用户数据的 split 分片信息，切片默认大小和 HDFS 的块大小相同（64MB），这样有利于 Map 任务的本地化执行，无须通过网络传递数据。这些信息包括 InputSplit 元数据信息和原始切分信息，其中元数据信息会被 JobTracker 使用，而原始切分信息会在 Map 任务初始化时用于获取自己要处理的数据信息，这两部分数据被保存到 job.split 和 job.splitmetainfo 文件中用于后续访问。切片信息由 InputSplit 对象封装，接口定义如下：

```
public interface InputSplit extends Writable {
    long getLength() throws IOException;
    String[] getLocations() throws IOException;
}
```

可以看到，split 并不包含具体的数据信息，而只包含数据的引用，Map 任务会根据引用地址去加载数据。InputSplit 是由 InputFormat 负责创建的：

```
public interface InputFormat<K,V> {
InputSplit[] getSplits(JobConf job,int numSplits) throws IOException;
RecordReader<K,V> getRecordReader(InputSplit split,JobConf job,
Reporter reporter) throws IOException;
}
```

4）提交作业到 JobTracker

JobClient 通过远程过程调用协议（RPC）将作业提交到 JobTracker 作业调度器中，首先为作业创建 JobInProgress 对象。JobTracker 会为用户提交的每一个作业创建一个 JobInProgress 对象，这个对象维护了作业的运行时信息，主要用于跟踪正在运行的作业的状态和进度。其次，检查用户是否具有指定队列的作业提交权限。Hadoop 作业调度以队列为单位来管理作业和资源，每个队列分配有一定的资源，调度器可以为每个队列指定哪些用户有权限提交作业。再次，检查作业配置的内存使用量是否合理。用户在提交作业时，可以分别通过参数 mapred.job.map.memory.mb 和 mapred.job.reduce.memory.mb 指定 Map Task 和 Reduce Task 的内存使用量，而管理员可以为集群中的 Map Task 和 Reduce Task 分别设置内存使用量，一旦用户配置的内存使用量超过总的内存限制，作业就会提交失败。最后，通知 TaskScheduler 初始化作业。JobTracker 收到提交的作业后，会交给

TaskScheduler 调度器，然后按照一定的策略对作业执行初始化操作。

2. 初始化作业

在完成 MapReduce 作业的提交之后，就开始作业初始化。初始化主要是指构造 Map Task 和 Reduce Task 并对它们进行初始化操作，这一步操作主要由调度器调用 JobTracker.initJob() 方法来执行。具体情况是 Hadoop 将每个作业分成 4 种类型的任务：Setup Task、Map Task、Reduce Task 和 Cleanup Task，它们的运行时信息由 TaskInProgress 维护，因此，创建这些任务就是创建 TaskInProgress 对象。初始化流程如图 2-6 所示。

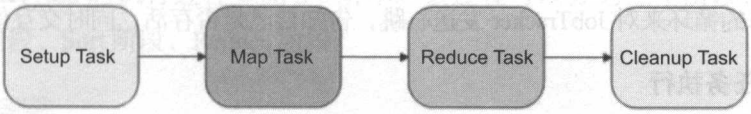


图 2-6　初始化流程图

从图 2-6 中可以看到作业初始化的 4 个主要流程。Setup Task 是作业初始化的标志性任务，它进行一些很简单的作业初始化工作，这个类型的任务分为 Map Setup Task 和 Reduce Setup Task 两种，并且只能运行一次。Map Task 是 Map 阶段的数据处理任务。Reduce Task 是 Reduce 阶段的数据处理任务，其数目可以由用户通过参数 mapred.reduce.tasks 指定。Hadoop 刚开始的时候只会调度 Map Task，直到 Map Task 完成数目达到由参数 mapred.reduce.slowstart.completed.maps 指定的百分比后，才开始调度 Reduce Task。Cleanup Task 是作业结束的标志性任务，主要做一些作业清理的工作，比如删除作业在运行中产生的一些临时目录和数据等信息。

3. 任务分配

JobTracker 节点遍历每一个 InputSplit，使用配置的任务调度器 TaskScheduler 来为某一个具体的 TaskTracker 节点分配任务，同时这个任务调度器只能决定给该 TaskTracker 节点分配哪一个 Job 或者哪些 Job 的任务以及分配多少个任务，而不能决定给当前的 TaskTracker 节点分配一个 Job 的具体哪一个任务。另外，针对一个具体的 TaskTracker 节点而言，任何一个作业都可以判断它的哪些 Map 任务相对于该 TaskTracker 节点来说属于本地任务，哪些 Map 任务属于非本地任务，根据其记录的引用地址选择距离最近的 TaskTracker 去执行，理想情况下切片信息就在 TaskTracker 的本地，这样就节省了网络数据传输的时间。当然，对于 Reduce 任务

来说，是没有本地任务与非本地任务这一说法的。因此，具体来讲，当任务调度器决定为一个 TaskTracker 节点分配一个 Job 的本地任务时，它会调用该 JobInProgress 对象的 obtainNewLocalMapTask() 方法；当分配一个非本地任务时，它会调用对应的 obtainNewNonLocalMapTask() 方法。那么以这个 TaskTracker 节点在集群中的物理位置为参考，这个 Job 可能有多个本地任务和多个非本地任务，至于为该 TaskTracker 节点分配哪一个本地或者非本地任务就由 JobInProgress 来决定了。当任务调度器为 TaskTracker 节点分配一个 Job 的 Reduce 任务时，就会调用该 Job 对应的 JobInProgress 对象的 obtainNewReduceTask() 方法。TaskTracker 会分配适当的固定数量的任务槽，理想状态一般遵循数据本地化和机架本地化。另外，TaskTracker 运行简单的循环来对 JobTracker 发送心跳，告知自己是否存活，同时交互信息。

4. 任务执行

TaskTracker 接到任务后开始执行如下操作：

（1）将任务 JAR 包从 HDFS 复制到本地。

（2）TaskTracker 新建本地目录，将 JAR 文件解压到此目录。

（3）TaskTracker 新建一个 TaskRunner 实例运行该任务。这样做的好处是即使所执行的任务出现了异常，也不会影响 TaskTracker 的运行使用。

1）MapTask

一个 MapTask 的运行过程如图 2-7 所示。运行过程可以分为三个阶段，分别是数据读取阶段、Map 阶段、分区阶段。

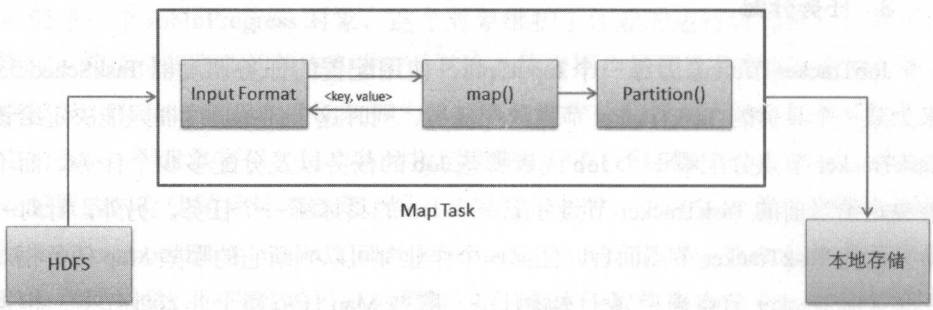


图 2-7　MapTask 的运行过程

数据读取阶段，是指 InputFormat 负责理解数据格式并解析成 <key, value> 传

递给 map 函数处理。InputFormat 提供了 getInputSplit 接口，负责将输入数据切分为一个个 InputSplit（分片），同时还提供了 getRecordReader 接口，负责解析 InputSplit，通过 RecordReader 提取出 key/value 键值对。这些键值对就作为 map 函数的输入。Map 阶段，是指运行用户实现的 Mapper 接口中的 map 函数，该函数负责对输入的 key/value 键值对进行初步处理，产生新的 key/value 键值对作为中间数据保存在本地磁盘上。分区阶段，map 函数输出的临时数据会被分成若干个分区，每个分区都会被一个 ReduceTask 处理。

2）ReduceTask

ReduceTask 的执行过程如图 2-8 所示。该过程可以分为三个阶段，分别为 Shuffle 阶段、Sort 阶段、Reduce 阶段。

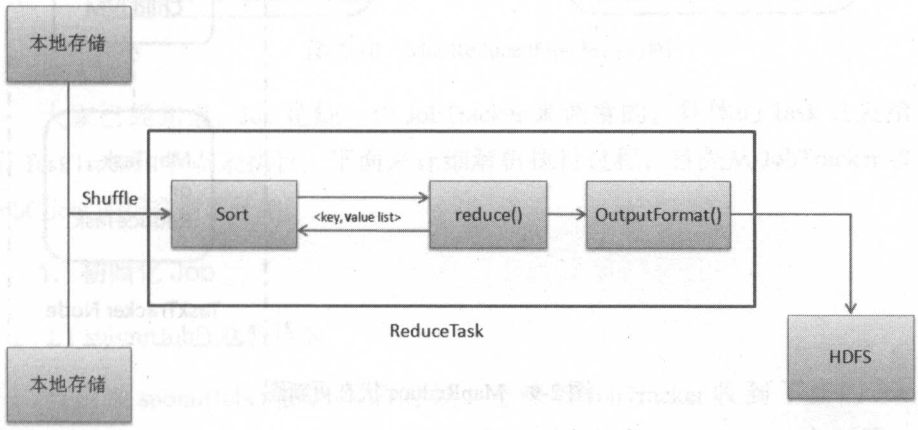


图 2-8　ReduceTask 的执行过程

Shuffle 阶段，是指 ReduceTask 通过查询 JobTracker 知道有哪些 Map Task 处理完毕。一旦有 MapTask 处理完毕，就会远程读取这些 MapTask 产生的临时数据。Sort 阶段，是指将读取的数据按照 key 对 key/value 键值对进行排序。Reduce 阶段，是指运行用户实现的 Reducer 接口中的 reduce 函数，该函数读取 Sort 阶段的有序 key/value 键值对进行处理，并将最终的处理结果存储到 HDFS 上。

5. 进程和状态的更新

MapReduce 的执行是一个漫长的过程，执行期间会将任务的进度反馈给用户。通过 Job 的 Status 属性对 Job 进行检测，例如作业状态、Map 和 Reduce 运行的进度、

Job 计数器的值、状态消息描述等，尤其是对计数器 Counter 属性的检查。状态更新在 MapReduce 系统中的传递流程如图 2-9 所示。

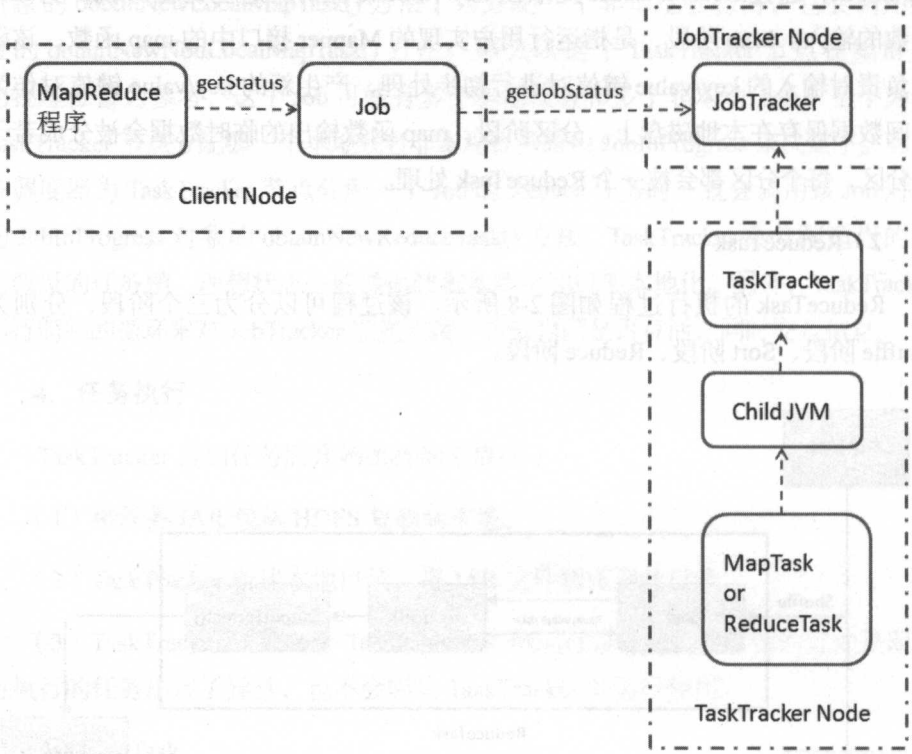


图 2-9　MapReduce 状态更新图

6. 作业的完成

当 JobTracker 收到 Job 最后一个 Task 完成的消息时，便把 Job 的状态设置为“完成”。JobClient 得知后，从 runJob() 方法返回，并清理 MapReduce 本地存储（mapred.local.dir 属性指定的目录），清理 Map 任务的输出文件。

2.2.3　JobTracker解剖

为了展示 JobTracker 的重要性，我们用另外一种方式——MapReduce 的运行架构展示出来，如图 2-10 所示。从图 2-10 中一眼就可以看出，JobTracker 在整个过程中发挥着举足轻重的作用。如果说 MapReduce 是 Hadoop 的“心脏”，那 JobTracker 就是 MapReduce 的“大脑”，控制着整个 MapReduce 过程。

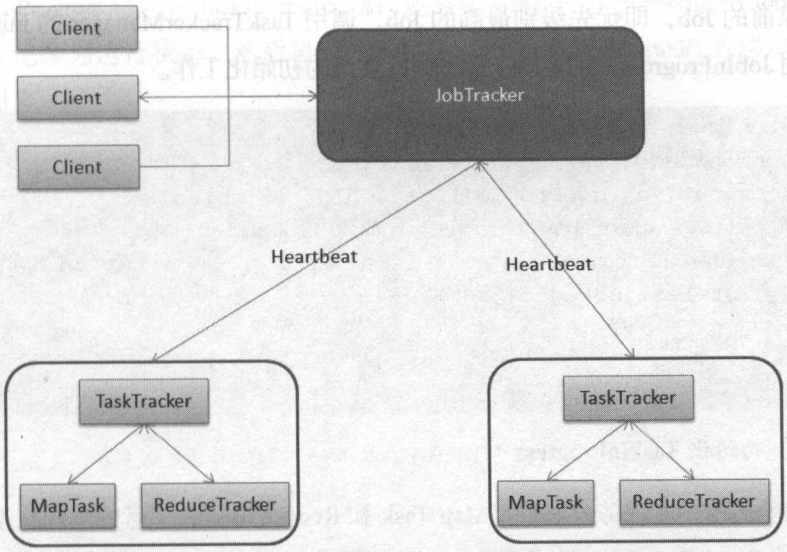


图 2-10　MapReduce 的简易运行图

大家已经知道，Job 是统一由 JobTracker 来调度的，具体的 Task 分发给各个 TaskTracker 节点来执行。下面来详细解析执行过程，首先从 JobTracker 收到 JobClient 的提交请求开始。

1. 初始化 Job

1）submitJob() 收到请求

当函数 submitJob() 被调用的时候，意味着 JobTracker 收到了新的 Job 请求。在收到新的 Job 请求之后，为了管理和调度任务，JobTracker 会创建一个 JobInProgress 对象。JobInProgress 在创建的时候会初始化一系列与任务有关的参数，并把在 JobClient 端上传的所有任务文件下载到本地文件系统中的临时目录里。这其中包括上传的应用的 JAR 文件包、记录配置信息的 configuration 文件、记录分割信息的文件等。

2）JobInitThread 通知初始化线程

JobTracker 使用 jobAdded() 方法将 Job 加入到需要初始化的队列里，即一个 list 成员变量 jobInitQueue 里。这个队列由 JobTracker 的监听器来维护，然后由 resortInitQueue() 方法根据作业的优先级排序。接着调用 notifyAll() 函数，会唤起一个用于初始化 Job 的线程 JobInitThread 来处理。JobInitThread 收到信号后即取

出最靠前的 Job，即优先级别最高的 Job，调用 TaskTrackerManager 的 initJob，最终调用 JobInProgress.initTasks() 方法执行真正的初始化工作。

```
synchronized (jobs) {
  synchronized (taskScheduler) {
    jobs.put(job.getProfile().getJobID(), job);
    //对JobTracker的每一个listener都调用jobAdded()函数
    for (JobInProgressListener listener : jobInProgressListeners) {
      listener.jobAdded(job);
    }
  }
}
```

3）初始化 TaskInProgress

如前文所述，任务分两种：Map Task 和 Reduce Task，二者的管理对象是相同的，都是 TaskInProgress。JobInProgress 会分别为 Map 和 Reduce 创建两个监控对象。接下来，我们对初始化 TaskInProgress 过程进行详细的分析。

首先，JobInProgress 会创建 Map 的监控对象。在 initTasks() 函数里通过调用 JobClient 的 readSplitFile() 方法获得已分解的输入数据的 RawSplit 列表，然后根据这个列表创建对应数目的 Map 执行管理对象 TaskInProgress。在这个过程中，还会记录该 RawSplit 块对应的所有在 HDFS 里的 blocks 所在的 DataNode 节点的 host，这个会在 RawSplit 创建时通过 FileSplit 的 getLocations() 函数获取，该函数会调用 DistributedFileSystem 的 getFileCacheHints() 获得。当然，如果存储在本地文件系统中，即使用 LocalFileSystem 时当然只有一个 location 即“localhost”了。

创建这些 TaskInProgress 对象完毕后，initTasks() 方法需要为这些 TaskInProgress 对象产生一个未执行任务的 Map 缓存 nonRunningMapCache，这一步是通过 createCache() 方法来完成的。这个 cache 产生之后，当 slave 端的 TaskTracker 向 master 发送心跳时，便可从此 cache 中提取需要执行的任务。

其次，JobInProgress 还会创建 Reduce 的监控对象。相对来说，这个过程较为简单。根据 JobConf 里指定的 Reduce 数目创建，默认情况下，只创建一个 Reduce 任务。同时 TaskInProgress 类用来监控和调度 Reduce 任务。initTasks() 也会通过 createCache() 方法产生 nonRunningReduceCache 成员。TaskInProgress 会根据不同的参数分别创建具体的 MapTask 或者 ReduceTask。

JobInProgress 创建完 TaskInProgress 后，还需要关键的一步——构造出 JobStatus，并将 Job 记录为运行状态。然后再调用 JobHistory.JobInfo.logStarted() 方法记录 Job 的执行日志。

至此，JobTracker 初始化 Job 的过程就全部完成了。具体代码如下：

```
public synchronized void initTasks() throws IOException {
  …
  //从HDFS中读取job.split文件从而生成Input Splits
  String jobFile = profile.getJobFile();
  Path sysDir = new Path(this.jobtracker.getSystemDir());
  FileSystem fs = sysDir.getFileSystem(conf);
  DataInputStream splitFile =
    fs.open(new Path(conf.get(«mapred.job.split.file»)));
  JobClient.RawSplit[] splits;
  try {
    splits = JobClient.readSplitFile(splitFile);
  } finally {
    splitFile.close();
  }
  //Map Task的个数就是Input Split的个数
  numMapTasks = splits.length;
  //为每个Map Task生成一个TaskInProgress来处理一个Input Split
  maps = new TaskInProgress[numMapTasks];
  for(int i=0; i < numMapTasks; ++i) {
    inputLength += splits[i].getDataLength();
    maps[i] = new TaskInProgress(jobId, jobFile,
                                 splits[i],
                                 jobtracker, conf, this, i);
  }
   //对于Map Task，将其放入nonRunningMapCache，是一个Map<Node,
List<TaskInProgress>>，也即对于Map Task来讲，其将会被分配到其Input Split
所在的节点上。nonRunningMapCache将在JobTracker向TaskTracker分配Map Task
的时候使用
  if (numMapTasks > 0) {
 nonRunningMapCache = createCache(splits, maxLevel);
}

  //创建Reduce Task
  this.reduces = new TaskInProgress[numReduceTasks];
  for (int i = 0; i < numReduceTasks; i++) {
    reduces[i] = new TaskInProgress(jobId, jobFile,
                                    numMapTasks, i,
```

```
                                        jobtracker, conf, this);
      //将Reduce Task放入nonRunningReduces，其将在JobTracker向
TaskTracker分配Reduce Task的时候使用
    nonRunningReduces.add(reduces[i]);
  }

  //创建两个Cleanup Task，一个用来清理Map，一个用来清理Reduce
  cleanup = new TaskInProgress[2];
  cleanup[0] = new TaskInProgress(jobId, jobFile, splits[0],
          jobtracker, conf, this, numMapTasks);
  cleanup[0].setJobCleanupTask();
  cleanup[1] = new TaskInProgress(jobId, jobFile, numMapTasks,
                      numReduceTasks, jobtracker, conf, this);
  cleanup[1].setJobCleanupTask();
  //创建两个初始化Task，一个初始化Map，一个初始化Reduce
  setup = new TaskInProgress[2];
  setup[0] = new TaskInProgress(jobId, jobFile, splits[0],
          jobtracker, conf, this, numMapTasks + 1 );
  setup[0].setJobSetupTask();
  setup[1] = new TaskInProgress(jobId, jobFile, numMapTasks,
                      numReduceTasks + 1, jobtracker, conf, this);
  setup[1].setJobSetupTask();
  tasksInited.set(true);//初始化完毕
  …
}
```

2. 调度 Job

Hadoop 内置默认的调度器 JobQueueTaskScheduler，其采用的策略是 FIFO（First Input First Output），即先进先出。需要特别说明的是，这种调度策略只是 Hadoop 原有的调度算法，随着 Hadoop 应用场合的增多，调度的策略也呈多样化。我们后续还会对各种调度的策略进行分析。

在调用 taskScheduler.start() 的过程中，为 JobTracker 注册了两个 Listener：EagerTaskInitializationListener 和 JobQueueJobInProgressListener。

- EagerTaskInitializationListener 是 JobTracker 中的监听器类，负责任务的初始化工作。EagerTaskInitializationListener 中有一个线程 JobInitThread，不断得到 jobInitQueue 中的 JobInProgress 对象，调用 JobInProgress 对象的 initTasks() 函数对任务进行初始化操作。

- JobQueueJobInProgressListener 是 JobTracker 中的另一个监听器类，用于监控 Job 的运行状态。它包含了一个映射，用来管理和调度所有的 JobInProgress。jobAdded() 函数会加入 Job 到 JobQueueJobInProgressListener 中的映射。

JobQueueTaskScheduler 中最重要的方法是 assignTasks，assignTasks 是用来分派任务的，那么分派过程是怎样的呢？具体实现：JobTracker 会检测 TaskTracker 的心跳，接到 TaskTracker 的 heartbeat() 调用后，首先会检查上一个心跳响应是否完成，是否要求启动或重启任务。如果上一个心跳已经响应，并且没有特殊的情况反馈回来，则表示一切正常，这时候 TaskTracker 就会处理本次心跳。

在分配 MapTask 或者 ReduceTask 之前，JobTracker 会检查 TaskTracker 端还可以做多少个 Map 和 Reduce 任务，将要派发的任务数是否超出这个数，是否超出集群的任务平均承载的负载数。如果都没有超过，就会为 TaskTracker 分配一个 MapTask 或 ReduceTask 。使用 JobInProgress 的 obtainNewMapTask() 方法来产生 MapTask，实质上最后调用了 JobInProgress 的 findNewMapTask() 方法访问 nonRunningMapCache。

在讲解任务初始化时我们说过，createCache() 方法会在网络拓扑结构上挂上需要执行的 TaskInProgress。findNewMapTask() 遵循的是就近原则，由近及远一层一层地寻找，首先是同一节点，然后再寻找同一机柜上的节点，接着寻找相同数据中心下的节点，直到找到最远层结束。这样，在 JobTracker 给 TaskTracker 派发任务的时候，可以迅速找到逻辑位置最近的 TaskTracker，然后让它执行任务。最终生成一个 Task 类对象，该对象被封装在一个 LanuchTaskAction 中，发回给 TaskTracker，让它去执行任务。

产生 Reduce 任务的过程类似，使用的是 JobInProgress.obtainNewReduceTask() 方法，实质上最后调用了 JobInProgress 的 findNewReduceTask() 方法访问 nonRuningReduceCache。具体代码如下：

```
public synchronized HeartbeatResponse heartbeat(TaskTrackerStatus
status,boolean initialContact,boolean acceptNewTasks,short responseId)
  throws IOException {
  ...
  String trackerName = status.getTrackerName();
  ...
  short newResponseId = (short)(responseId + 1);
```

```
    ...
    HeartbeatResponse response = new HeartbeatResponse(newResponseId, null);
    List<TaskTrackerAction> actions = new ArrayList<TaskTrackerAction>();
    //如果TaskTracker向JobTracker请求一个Task运行
    if (acceptNewTasks) {
      TaskTrackerStatus taskTrackerStatus = getTaskTracker(trackerName);
      if (taskTrackerStatus == null) {
        LOG.warn("Unknown task tracker polling; ignoring:"+ trackerName);
      } else {
        //setup和cleanup的Task优先级最高
        List<Task> tasks = getSetupAndCleanupTasks(taskTrackerStatus);
        if (tasks == null ) {
          //任务调度器分配任务
          tasks = taskScheduler.assignTasks(taskTrackerStatus);
        }
        if (tasks != null) {
          for (Task task : tasks) {
            //将任务放入actions列表，返回给TaskTracker
            expireLaunchingTasks.addNewTask(task.getTaskID());
            actions.add(new LaunchTaskAction(task));
          }
        }
      }
    }
    ...
    int nextInterval = getNextHeartbeatInterval();
    response.setHeartbeatInterval(nextInterval);
    response.setActions(
                        actions.toArray(new TaskTrackerAction[actions.
size()]));
    ...
    return response;
  }

  public synchronized List<Task> assignTasks(TaskTrackerStatus taskTracker)
      throws IOException {
    ClusterStatus clusterStatus = taskTrackerManager.getClusterStatus();
    int numTaskTrackers = clusterStatus.getTaskTrackers();
    Collection<JobInProgress> jobQueue = jobQueueJobInProgressListener.
getJobQueue();
    int maxCurrentMapTasks = taskTracker.getMaxMapTasks();
    int maxCurrentReduceTasks = taskTracker.getMaxReduceTasks();
    int numMaps = taskTracker.countMapTasks();
```

```
int numReduces = taskTracker.countReduceTasks();
//计算剩余的Map和Reduce的工作量：remaining
int remainingReduceLoad = 0;
int remainingMapLoad = 0;
synchronized (jobQueue) {
  for (JobInProgress job : jobQueue) {
    if (job.getStatus().getRunState() == JobStatus.RUNNING) {
      int totalMapTasks = job.desiredMaps();
      int totalReduceTasks = job.desiredReduces();
      remainingMapLoad += (totalMapTasks - job.finishedMaps());
      remainingReduceLoad += (totalReduceTasks - job.finishedReduces());
    }
  }
}
//计算平均每个TaskTracker应有的工作量，remaining/numTaskTrackers是剩余的工作量除以TaskTracker的个数
int maxMapLoad = 0;
int maxReduceLoad = 0;
if (numTaskTrackers > 0) {
  maxMapLoad = Math.min(maxCurrentMapTasks,
                        (int) Math.ceil((double) remainingMapLoad /
                                        numTaskTrackers));
  maxReduceLoad = Math.min(maxCurrentReduceTasks,
                           (int) Math.ceil((double) remainingReduceLoad
                                           / numTaskTrackers));
}
...

//Map优先于Reduce，当TaskTracker上运行的Map Task数目小于平均的工作量时，则向其分配Map Task
if (numMaps < maxMapLoad) {
  int totalNeededMaps = 0;
  synchronized (jobQueue) {
    for (JobInProgress job : jobQueue) {
      if (job.getStatus().getRunState() != JobStatus.RUNNING) {
        continue;
      }
      Task t = job.obtainNewMapTask(taskTracker, numTaskTrackers,
          taskTrackerManager.getNumberOfUniqueHosts());
      if (t != null) {
        return Collections.singletonList(t);
      }
      ...
```

```
      }
    }
  }
  //分配完Map Task，再分配Reduce Task
  if (numReduces < maxReduceLoad) {
    int totalNeededReduces = 0;
    synchronized (jobQueue) {
      for (JobInProgress job : jobQueue) {
        if (job.getStatus().getRunState() != JobStatus.RUNNING ||
            job.numReduceTasks == 0) {
          continue;
        }
        Task t = job.obtainNewReduceTask(taskTracker, numTaskTrackers,
            taskTrackerManager.getNumberOfUniqueHosts());
        if (t != null) {
          return Collections.singletonList(t);
        }
        ...
      }
    }
  }
  return null;
}
```

2.2.4 TaskTracker 解剖

1. 加载 Task

Task 的执行实际是由 TaskTracker 发起的，TaskTracker 会定期与 JobTracker 进行一次通信，报告自己 Task 的执行状态，接收 JobTracker 的指令等。如果发现有自己需要执行的新任务也会在这时启动，即在 TaskTracker 调用 JobTracker 的 heartbeat() 方法时进行，此调用底层是通过 IPC 层调用 Proxy 接口实现的。下面简单介绍一下每个步骤。

1）连接 JobTracker

TaskTracker 的启动过程会初始化一系列参数和服务，然后尝试连接 JobTracker。如果连接断开，则会循环尝试连接 JobTracker，并重新初始化所有成员和参数。

如果连接 JobTracker 服务成功，TaskTracker 就会进入到一个循环中。这个循环会每隔 10 秒（可定义）与 JobTracker 通信一次，调用 transmitHeartBeat() 函数

获得 HeartbeatResponse 信息。然后调用 HeartbeatResponse 的 getActions() 函数获得 JobTracker 传过来的所有指令，即一个 TaskTrackerAction 数组。接着遍历这个数组，如果是一个新任务指令即 LaunchTaskAction，则调用 addToTaskQueue 加入到待执行队列；否则加入到 tasksToCleanup 队列，交给一个 taskCleanupThread 线程来处理，如执行 KillJobAction 或者 KillTaskAction 等。

2）获取 JobTracker 指令

在 transmitHeartBeat() 函数处理中，TaskTracker 会创建一个新的 TaskTrackerStatus 对象记录目前任务的执行状况，检查目前执行的 Task 数目及本地磁盘的空间使用情况等，如果可以接收新的 Task，则设置 heartbeat() 的 askForNewTask 参数为 true。然后通过 IPC 接口调用 JobTracker 的 heartbeat() 方法发送过去，heartbeat() 返回值是 TaskTrackerAction 数组。

3）处理新任务

TaskLauncher 是用来处理新任务的线程类，包含了一个待运行任务的队列 tasksToLaunch。TaskTracker.addToTaskQueue 会调用 TaskTracker 的 registerTask，创建 TaskInProgress 对象来调度和监控任务，并把它加入到 runningTasks 队列中。同时将这个 TaskInProgress 加入到 tasksToLaunch 中，并调用 notifyAll() 方法唤醒一个线程运行。该线程从队列 tasksToLaunch 中取出一个待运行任务，调用 TaskTracker 的 startNewTask 运行任务。

TaskTracker.startNewTask() 启动新任务，调用 localizeJob() 真正初始化 Task 并开始执行。TaskTracker.localizeJob() 函数用来初始化 Job 目录等。此函数的主要任务是初始化工作目录 workDir，再将应用程序的 JAR 包从 HDFS 复制到本地文件系统中，调用 RunJar.unJar() 函数将包解压到工作目录。然后创建一个 runningJob 并调用 addTaskToJob() 函数将它添加到 runningJobs 监控队列中。addTaskToJob() 方法把一个任务加入到该任务属于的 runningJob 的 Tasks 列表中。如果该任务属于的 runningJob 不存在，则先新建，再加入到 runningJobs 中。完成后即可调用 launchTaskForJob() 方法开始执行 Task。

4）执行任务

启动 Task 的工作实际是调用 TaskInProgress 的 launchTask() 函数来执行的。执行任务前先调用 localizeTask() 函数更新一下 jobConf 文件并写入到本地目录中，

然后通过调用Task的createRunner()方法创建TaskRunner对象并调用其start()方法，最后启动Task独立的Java执行子进程。

然后由Task.createRunner()创建启动Runner对象。Task有两个实现版本，即MapTask和ReduceTask，分别用于创建Map和Reduce任务。MapTask会创建MapTaskRunner来启动Task子进程，而ReduceTask则创建ReduceTaskRunner来启动。

完成创建之后，将由TaskRunner.start()启动子进程。TaskRunner负责将一个任务放到一个进程里面来执行。它会调用run()函数来处理，主要工作就是初始化启动Java子进程的一系列环境变量，包括设定工作目录workDir、设置CLASSPATH环境变量等。然后装载应用程序的JAR包。JvmManager用于管理该TaskTracker上所有运行的Task子进程。每一个进程都是由JvmRunner来管理的，它也是位于单独线程中的。JvmManager的launchJvm()方法，根据任务是Map还是Reduce，生成对应的JvmRunner并放到对应JvmManagerForType的进程容器中进行管理。JvmManagerForType的reapJvm()分配一个新的JVM进程。如果JvmManagerForType槽满，就寻找idle的进程，如果是同Job的则直接放进去，否则杀死这个进程，用一个新的进程代替；如果槽没有满，那么就启动新的子进程。生成新的进程使用spawnNewJvm()方法。spawnNewJvm()使用JvmRunner线程的run()方法，run()方法用于生成一个新的进程并运行它，具体实现是调用runChild。

2. MapTask的执行

MapTask的实际执行载体是Child，它包含一个main()函数，进程执行，会将相关参数传递进来，它会拆解这些参数，通过getTask(jvmId)向父进程索取任务，并且构造出相关的Task实例，然后使用Task的run()方法启动任务。

run()方法相当简单，配置完系统的TaskReporter后，就根据情况执行runJobCleanupTask、runJobSetupTask、runTaskCleanupTask或Mapper。由于MapReduce现在有两套API，因此MapTask需要支持这两套API，使得MapTask执行Mapper分为runNewMapper和runOldMapper。下面我们分析一下runOldMapper。

runOldMapper最开始部分是构造Mapper处理的InputSplit，然后开始创建Mapper的RecordReader，最终得到Map的输入。之后构造Mapper的输出，是通过MapOutputCollector进行的，也分两种情况：如果没有Reducer，则用

DirectMapOutputCollector；否则，用 MapOutputBuffer。

构造完 Mapper 的输入 / 输出，通过构造配置文件中配置的 MapRunnable，就可以执行 Mapper 了。目前系统有两个 MapRunnable：MapRunner 和 MultithreadedMapRunner。MapRunner 是单线程执行器，比较简单，它会使用反射机制生成用户定义的 Mapper 接口实现类作为它的一个成员。

MapRunner 的 run() 方法会先创建对应的 <key,value> 对象，然后对 InputSplit 的每一对调用用户实现的 Mapper 接口实现类的 map() 方法。每处理一个数据对，就要使用 OutputCollector 收集每次处理 key/value 对后得到的新 key/value 对，把它们存储（spill）到文件或者放到内存，以做进一步的处理，比如排序、combine 等。

MapOutputCollector 有两个子类：MapOutputBuffer 和 DirectMapOutputCollector。DirectMapOutputCollector 用在不需要 Reduce 阶段的时候。如果 Mapper 后续有 Reduce 任务，则系统会使用 MapOutputBuffer 作为输出。MapOutputBuffer 使用了一个缓冲区对 Map 的处理结果进行缓存并放在内存中，又使用几个数组对这个缓冲区进行管理。

SpillThread 线程将缓冲区中的数据 spill 到硬盘中。接下来，我们对 SpillThread 进行简单的分析。

在适当的时机，缓冲区中的数据会被 spill 到硬盘中。首先，为什么需要 spill 呢？因为内存的大小总是有限的，在 Mapper 处理过程中，数据持续输出到内存中时，必然需要有机制能将内存中的数据换出，合理地输出到磁盘上。SpillThread 就是用来完成这部分工作的。SpillThread 的线程处理函数只是做一层封装，当索引表中的 kvstart 和 kvend 指向相同的索引位置时，会持续处于等待过程，等待外部通知需要触发 spill 动作。当有 spill 请求时，会触发 StartSpill 来唤醒 SpillThread 线程，进入 sortAndSpill。如下代码就是 SpillThread 线程体函数。

```
protected class SpillThread extends Thread {
  @Override
  public void run() {
    spillLock.lock();
    spillThreadRunning = true;
    try {
      while (true) {
        spillDone.signal();
```

```
          while (kvstart == kvend) {
// 等待被唤醒
            spillReady.await();
          }
          try {
            spillLock.unlock();
// spill处理
            sortAndSpill();
          } catch (...) {
...
          } finally {
            spillLock.lock();
// 重置索引区，更新buf缓冲区的尾部位置信息
            if (bufend < bufindex && bufindex < bufstart) {
              bufvoid = kvbuffer.length;
            }
            kvstart = kvend;
            bufstart = bufend;
          }
        }
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      } finally {
        spillLock.unlock();
        spillThreadRunning = false;
      }
    }
  }
```

线程函数内的处理逻辑比较简单，主要分为三个步骤：

（1）等待唤醒。

（2）对内存中的数据进行排序并将数据溢出写入到磁盘中。

（3）重置索引区和缓存区的 end 标记。

3. ReduceTask的执行

ReduceTask.run() 方法一开始和 MapTask 类似，包括 initialize() 初始化、runJobCleanupTask()、runJobSetupTask()、runTaskCleanupTask()。之后进入正式的工作，主要包括三个步骤：Copy、Sort、Reduce。Copy 就是远程复制 Map 的输出数据，Sort 就是对所有的数据进行排序，Reduce 做聚集就是我们自己写的 Reducer。和 MapTask

类似，Child 类中直接运行的是 run() 方法。ReduceTask.run() 方法的代码如下：

```
@Override
@SuppressWarnings("unchecked")
public void run(JobConf job, final TaskUmbilicalProtocol umli
      throws IOException, InterruptedException, ClsNotFoundException {
      this.umbilical = umbilical;
      job.setBoolean("mapred.skip.on", isSkipping());
      /*添加Reduce过程需要经过的几个阶段，以便通知TaskTracker目前运行的情况*/
   if (isMapOrReduce()) {
   copyPhase = getProgress().addPhase("copy");
   sortPhase  = getProgress().addPhase("sort");
   reducePhase = getProgress().addPhase("reduce");
 }
 // start thread that will handle communication with parent
 /设置并启动Reporter进程以便和TaskTracker进行交流
 TaskReporter reporter = new TaskReporter(getProgress(), umlical,
     jvmContext);
 reporter.startCommunicationThread();
 //在Job Client中初始化Job时，默认使用新的API，详见etUseNewAPI()
 boolean useNewApi = job.getUseNewReducer();
 /*用来初始化任务，主要进行一些和任务输出相关的设置，比如创建ter、工作目录等*/
 initialize(job, getJobID(), reporter, useNewApi);
     /*以下4条if语句均根据任务类型的不同进行相应的操作，这些方法是Task类的方
法，所以与任务是MapTask还是ReduceTask无关*/
     // check if it is a cleanupJobTask
     if (jobCleanup) {
       runJobCleanupTask(umbilical, reporter);
       return;
     }
     if (jobSetup) {
         //主要创建工作目录的FileSystem对象
       runJobSetupTask(umbilical, reporter);
       return;
     }
 if (taskCleanup) {
       //设置任务目前所处的阶段为结束阶段，并且删除工作目录
   runTaskCleanupTask(umbilical, reporter);
   return;
 }

 // Initialize the codec
 codec = initCodec();
```

```
//判断是否是单机op
 boolean isLocal="locequals(job.get("mapred.job.tracker", "lal"));
 if (!isLocal) {
      //1.Copy.就是从执行各个Map任务的服务器那里收到Map的输出。拷贝的任务是
由ReduceTask.ReduceCopier类来负责的
       //ReduceCopier对象负责将Map函数的输出复制到Reduce所在机器
    reduceCopier = new ReduceCopier(umbilical, job, reporter);
    if (!reduceCopier.fetchOutputs()) {////fetchOutputs函数负责复制Map函数的输出
     if(reduceCopier.mergeThrowable instanceof FSError) {
       throw (FSError)reduceCopier.mergeThrowable;
     }
     throw new IOException("Task: " + getTaskID() +
         " - The reduce copier failed", receer.mergeThrowable);
   }
 }
 copyPhase.complete();                          // 复制已经完成
 setPhase(TaskStatus.Phase.SORT);
 statusUpdate(umbilical);

 final FileSystem rfs = FileSystem.getLocal(job).getRaw();
 //2.Sort(其实相当于合并).排序工作，就相当于上述复制工作的一个延续。它会在所有
文件都复制完毕后进行
 //使用工具类Merger归并所有的文件。经过这样一个流程，一个合并了所有Map任务输出
文件的新文件产生了
 //而那些从其他各个服务器网罗过来的Map任务输出文件全部被删除了

 //根据Hadoop是否分布式来决定调用哪种排序方式
 RawKeyValueIterator rIter = isLocal
   ? Merger.merge(job, rfs, job.getMapOutputKeyClass(),
       job.getMapOutputValueClass(), codec, getMapFiles(rfs, tr),
       !conf.getKeepFailedTaskFiles(), joge "io.sort.factor", 100),
       new Path(getTaskID().toString()), jogeutKeyComparator(),
       reporter, spilledRecordsCounter, null)
   : reduceCopier.createKVIterator(job, rfs, reporter);

 mapOutputFilesOnDisk.clear();

   sortPhase.complete();
   setPhase(TaskStatus.Phase.REDUCE);
   statusUpdate(umbilical);
    //3.Reduce 1.Reduce任务的最后一个阶段。它会准备好Map的 keyClass
("mapred.output.key.class"或"mapred.mapoutput.key.class")
   //valueClass("mapred.mapoutput.value.class"或"maputput.value.class")
```

```
    //和Comparator ("mapred.output.value.groupfn.class"或 "maoutput.key.
comparator.class")
    Class keyClass = job.getMapOutputKeyClass();
    Class valueClass = job.getMapOutputValueClass();
    RawComparator comparator = job.tputValueGroupingComparator();
    //2.根据参数useNewAPI判断执行runNewReduce还是runOuce。分析runNewReduce
    if (useNewApi) {
          //3.runNewReducer
          //0.向报告进程书写一些信息
            //1.获得一个TaskAttemptContext对象。通过这个对象，创建
reducerContext
        //2.reducer.run(reducerContext)开始执行Reduce
      runNewReducer(job, umbilical, reporter, rIter, compr,
                    keyClass, valueClass);
    } else {
      runOldReducer(job, umbilical, reporter, rIter, compr,
                    keyClass, valueClass);
    }
    done(umbilical, reporter);
  }
```

接下来我们对这个三个过程进行逐一分析。

1）Copy 过程

Copy 就是从执行各个 Map 任务的服务器那里收集 Map 的输出文件。复制的任务由 ReduceTask.ReduceCopier 类负责。其大致流程为：

（1）索取任务。使用 GetMapEventsThread 线程，该线程的 run() 方法不停地调用 getMapCompletionEvents() 方法，该方法又使用 RPC 调用 TaskUmbilicalProtocol 协议的 getMapCompletionEvents，方法使用所属的 jobID 向其父 TaskTracker 询问此作业各 Map 任务的完成状况（TaskTracker 要向 JobTracker 询问后再转告给它），返回一个数组 TaskCompletionEventevents[]。TaskCompletionEvent 包含 taskID 和 IP 地址之类的信息。

（2）当获取到相关 Map 任务执行服务器的信息后，有一个线程 MapOutputCopier 开启，做具体的复制工作。它会在一个单独的线程内负责某个 Map 任务服务器上文件的复制工作。MapOutputCopier 的 run() 方法循环调用 copyOutput，copyOutput 又调用 getMapOutput，使用 HTTP 远程复制。

（3）getMapOutput 远程复制过来的内容（当然也可以是本地的）作为 MapOutput 对象存在，它可以在内存中，也可以序列化在磁盘上，可根据内存使用状况来自动调节。

（4）同时还有一个内存 Merger 线程 InMemFSMergeThread 和一个文件 Merger 线程 LocalFSMerger 同步工作，它们将下载过来的文件（可能在内存中，统称为文件）进行归并排序，以此节约时间，降低输入文件的数量，为后续的排序工作减负。InMemFSMergeThread 的 run() 方法循环调用 doInMemMerge，该方法使用工具类 Merger 实现归并。如果需要归并，则调用 combinerRunner.combine。

2）Sort 过程

排序工作，就相当于上述复制工作的一个延续。它会在所有的文件都复制完毕后进行。使用工具类 Merger 归并所有的文件。经过这样一个流程，一个合并了所有所需 Map 任务输出文件的新文件产生了。而那些从其他各个服务器收集过来的 Map 任务输出文件全部被删除了。

3）Reduce 过程

Reduce 任务的最后一个阶段。它会准备好 keyClass、valueClass 和 Comparator。其中 keyClass 为 mapred.output.key.class 或者 mapred.mapoutput.key.class；valueClass 为 mapred.mapoutput.value.class 或者 mapred.output.value.class；Comparator 为 mapred.output.value.groupfn.class 或者 mapred.output.key.comparator.class。

最后调用 runOldReducer() 方法，也是两套 API，这里我们分析 runOldReducer。

在输出方面，它会准备一个 OutputCollector 收集输出。与 MapTask 不同，这个 OutputCollector 更为简单，仅仅是打开一个 RecordWriter，collect 一次，write 一次。最大的不同在于，这次传入 RecordWriter 的文件系统基本都是分布式文件系统，或者说是 HDFS。

在输入方面，ReduceTask 会用准备好的 KeyClass、ValueClass、KeyComparator 等自定义类，构造出 Reducer 所需的键类型和值的迭代类型 Iterator（一个键到了这里一般对应一组值）。

有了输入，有了输出，不断循环调用自定义的 Reducer，最终 Reduce 阶段完成。

2.2.5 失败场景分析

在 MapReduce 作业执行过程中也会出现失败情况，以下对失败的几种情况进行分析。

1. Task 失败（Map 或者 Reduce）

第一种情况：Map 或 Reduce 任务中的用户代码抛出运行异常。

错误原因可能是用户代码出现异常，或者任务超过 mapred.task.timeout 指定的时间依然没有返回。

此时子进程 JVM 会在退出之前向 TaskTracker 发送错误报告，错误报告被记录在错误日志中。然后 JobTracker 会调度其他 TaskTracker 来重新执行此任务。如果失败次数超过 4 次（通过 mapred.map.max.attempts 和 mapred.reduce.max.attempts 属性来设置，默认为 4），则 Job 以失败告终。如果系统不想以这种方式结束退出，而是想通过 Task 成功数的百分比来决定 Job 是否通过，则可以指定如下两个属性：mapred.max.map.failures.percent（Map 任务最大失败率）和 mapred.max.reduce.failures.percent（Reduce 任务最大失败率）。如果失败比率超过指定的值，则 Job 以失败告终。

第二种情况：子进程 JVM 突然退出。TaskTracker 会注意到 JVM 退出，并将此 Task Attempt 标记为失败。JobTracker 通过心跳得知一个 Task Attempt 失败后，会重启调度该 Task 的执行。默认情况下如果失败 4 次不会重试（通过 mapred.map.max.attempts 可改变这个次数），则整个 Job 也会标记为失败。

2. TaskTracker 失败

如果 TaskTracker 由于崩溃或者运行过慢失败，则停止向 JobTracker 发送心跳，JobTracker 会注意到这点并将这个 TaskTracker 从等待任务调度 TaskTracker 池中移除。

即使 TaskTracker 没有失败，也有可能因为失败任务次数远远高于集群的平均失败次数而被列入黑名单，在重启后才将此 TaskTracker 移出黑名单。

3. JobTracker 失败

JobTracker 失败，如果出现问题也是最为严重的问题，此时只得重新开始提交运行。

2.3 MapReduce 实例分析

单词计数是最简单也是最能体现 MapReduce 思想的程序之一，可以称为

MapReduce 版“Hello World”，该程序的完整代码可以在 Hadoop 安装包的“src/examples”目录下找到。单词计数主要完成的功能是：统计一系列文本文件中每个单词出现的次数。

2.3.1 运行 WordCount 程序

1. 准备工作阶段

现在以“hadoop”普通用户登录“Master.Hadoop”服务器。

（1）创建本地示例文件。首先在“/home/hadoop”目录下创建文件夹“file”。接着创建两个文本文件 file1.txt 和 file2.txt，使 file1.txt 的内容为“Hello World”，而 file2.txt 的内容为“Hello Hadoop”。整个过程比较简单，就不在此展现命令了。

（2）在 HDFS 上创建输入文件夹，命令如下：

```
#hadoop fs -mkdir input
#hadoop fs -ls
```

（3）上传本地 file 中的文件到集群的 input 目录下，命令如下：

```
#hadoop fs -put file/file!.txt input
#hadoop fs -ls input
```

2. 运行阶段

在集群上运行 WordCount 程序（以 input 作为输入目录，output 作为输出目录）。已经编译好的 WordCount 的 JAR 在“/usr/hadoop”目录下，即“hadoop-examples-1.0.0.jar”，所以在下面执行命令时记得把路径写全了，否则会提示找不到该 JAR 包。MapReduce 执行过程显示如下信息：

```
#hadoop jar /usr/hadoop/hadoop-examples-1.0.0.jar wordcount input output
```

Hadoop 命令会启动一个 JVM 来运行这个 MapReduce 程序，并自动获得 Hadoop 的配置，同时把类的路径（及其依赖关系）加入到 Hadoop 的库中。

3. 查看结果

（1）查看 HDFS 上的 output 目录内容，命令如下：

```
#hadoop fs -ls output
```

（2）查看结果输出文件内容，命令如下：

```
#hadoop fs -cat output/part-r-00000
```

2.3.2 WordCount 源码分析

1. 源代码程序

```
package org.apache.hadoop.examples;
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
public class WordCount {
    public static class TokenizerMapper
           extends Mapper<Object, Text, Text, IntWritable>{
           private final static IntWritable one = new IntWritable(1);
           private Text word = new Text();

           public void map(Object key, Text value, Context context)
              throws IOException, InterruptedException {
              StringTokenizer itr = new StringTokenizer(value.toString());
              while (itr.hasMoreTokens()) {
              word.set(itr.nextToken());
              context.write(word, one);
           }
       }
    }
    public static class IntSumReducer
           extends Reducer<Text,IntWritable,Text,IntWritable> {
           private IntWritable result = new IntWritable();
           public void reduce(Text key, Iterable<IntWritable> values,Context
context)
                  throws IOException, InterruptedException {
              int sum = 0;
```

```
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).
getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("Usage: wordcount <in> <out>");
            System.exit(2);
        }
        Job job = new Job(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

2. Map 过程

```
public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
    }
}
```

Map 过程需要继承 org.apache.hadoop.mapreduce 包中的 Mapper 类，并重写其 map() 方法。通过在 map() 方法中添加两条把 key 值和 value 值输出到控制台的代码，可以发现 map() 方法中 value 值存储的是文本文件中的一行（以回车符为行结束标记），而 key 值为该行的首字母相对于文本文件首地址的偏移量。然后 StringTokenizer 类将每一行拆分成一个个的单词，并将 <word,1> 作为 map() 方法的结果输出，其余工作都交由 MapReduce 框架处理。

3. Reduce 过程

```
public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();
    public void reduce(Text key, Iterable<IntWritable>values,Context context)
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

Reduce 过程需要继承 org.apache.hadoop.mapreduce 包中的 Reducer 类，并重写其 reduce() 方法。Map 过程输出的 <key,values> 中 key 为单个单词，而 values 是对应单词的计数值所组成的列表。Map 的输出就是 Reduce 的输入，所以 reduce() 方法只要遍历 values 并求和，即可得到某个单词的总次数。

4. 执行 MapReduce 任务

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
        System.err.println("Usage: wordcount <in> <out>");
        System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
```

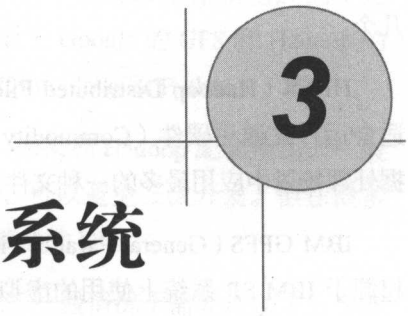
```
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

在 MapReduce 中，由 Job 对象负责管理和运行一个计算任务，并通过 Job 的一些方法对任务的参数进行相关设置。此处设置了使用 TokenizerMapper 完成 Map 过程中的处理及使用 IntSumReducer 完成 Combine 和 Reduce 过程中的处理。另外还设置了 Map 过程和 Reduce 过程的输出类型：key 的类型为 Text，value 的类型为 IntWritable。任务的输出和输入路径则由命令行参数指定，并由 FileInputFormat 和 FileOutputFormat 分别设定。完成相应任务的参数设定后，即可调用 job.waitForCompletion() 方法执行任务。

2.4 小结

MapReduce 作为 Hadoop 的“心脏”，本章对其进行了详细的分析。首先从设计思想入手，搞清楚 MapReduce 的精华在哪里。然后详细从 Code 层面阐述了 MapReduce 的作业执行流程，对“四大金刚”中的 JobTracker 和 TaskTracker 进行了详细的解剖。其实本章只是带领读者对经典的 MapReduce 框架进行了分析，随着每个版本的发布，有可能其中提到的函数和方法已经改变，甚至对某些流程也做出了调整。如果读者想深入了解，建议下载最新的源码进行分析。当然，这需要足够的耐心。

总之，要想拿下 Hadoop，必须先过 MapReduce 这一关。



分布式文件系统

3.1 群雄并起的 DFS

在计算机存储业发展之初，人们将数据存储在硬盘上，但这种状态并未持续太久，因为本地存储空间受限。随着互联网的出现和发展，又出现了一个问题，即在多个节点协同工作的时候，两个以上的节点不能在同一时间读 / 写和操作同一文件。逐渐地，人们试着将网络上的硬盘统一筹划管理来扩大存储空间，并解决同时读 / 写这一问题。最具代表性的是 SUN 公司提出的 NFS（Network File System），它的目的就是想让不同的机器、不同的作业系统可以彼此分享文件。文件系统最初设计时，仅仅是为局域网内的本地数据服务的。随后发展到分布式文件系统（Distributed File System），将服务范围扩展到了整个网络。

那么什么是分布式文件系统呢？所谓分布式，是指文件系统管理的物理存储资源不一定直接连接在本地节点上，而是通过计算机网络与节点相连。将固定于某个地点的某个文件系统扩展到任意多个地点 / 文件系统，众多节点组成一个文件系统网络。每个节点可以分布在不同的地点，通过网络进行节点间的通信和数据传输。人们在使用分布式文件系统时，无须关心数据是存储在哪个节点上的或者是从哪个节点获取的，只需要像使用本地文件系统一样管理和存储文件系统中的数据。

一时间，各大 IT 公司都设计并推出了自己的文件系统，尤其是硬件厂商，纷

纷推出了最适合自己生产的硬件性能的 DFS。比较流行的分布式文件系统有如下几个。

HDFS（Hadoop Distributed File System）：Hadoop 分布式文件系统，被设计成适合运行在通用硬件（Commodity Hardware）上的分布式文件系统。它是在大数据处理场景中应用最多的一种文件系统，后续章节会详细分析。

IBM GPFS（General Parallel File System）：是 IBM 公司第一个共享文件系统，起源于 IBM SP 系统上使用的虚拟共享磁盘技术（VSD）。作为这项技术的核心，GPFS 是一个并行的磁盘文件系统，它保证在资源组内的所有节点可以并行访问整个文件系统；而且针对此文件系统的服务操作，可以同时安全地在使用此文件系统的多个节点上实现。GPFS 允许客户共享文件，而这些文件可能分布在不同节点的不同硬盘上；它提供了许多标准的 UNIX 文件系统接口，允许应用不需要修改或者重新编辑就可以在其上运行。

GoogleFS：是一个可扩展的分布式文件系统，用于大型的、分布式的、对大量数据进行访问的应用。它运行于廉价的普通硬件上，但可以提供容错功能。它可以给大量的用户提供总体性能较高的服务。

MogileFS：是一套高效的文件自动备份组件，由 Six Apart 开发，广泛应用在 LiveJournal 等 Web 2.0 站点上。

OpenAFS：是一个分布式文件系统，有着卡耐基梅隆大学的支持。OpenAFS 提供有架构、文件共享客户端、服务器、具有只读形式的共享、单个客户端位置的独立性、可扩展、安全、对用户透明共享资源的能力，是一个比较成熟稳定的文件系统。

Lustre：是一个平行分布式文件系统，通常用于大型计算机集群和超级计算机上。Lustre 是源自 Linux 和 Cluster 的混成词。最早在 1999 年，由 Peter Braam 创建的集群文件系统公司（Cluster File Systems Inc.）开始研发，于 2003 年发布 Lustre 1.0 版本。它采用 GNU GPLv2 开源码授权。

FastDFS：是一个开源的轻量级分布式文件系统，它对文件进行管理，功能包括文件存储、文件同步、文件访问（文件上传、文件下载）等，解决了大容量存储和负载均衡的问题，特别适合以文件为载体的在线服务，如相册网站、视频网站等。

KFS（Kosmos Distributed File System）：是一个专门为数据密集型应用（搜索引擎、数据挖掘等）而设计的存储系统，类似于 Google 的 GFS 和 Hadoop 的 HDFS。KFS 使用 C++ 实现，支持的客户端包括 C++、Java 和 Python。

用户可以根据自己的需求选择 DFS 来使用。如果和 Hadoop 配套使用，一般情况下，笔者建议选择与其有较好融合的 HDFS，可以避免二次开发。但在很多情况下，受商业和安全等因素限制，也可以选择其他的 DFS。

那么，如何选择既合适又性能优秀的 DFS 呢？一般情况下需要从以下三个方面来决策：

（1）数据的读取速率，包括响应用户读取数据文件的请求、定位数据文件所在的节点、读取实际硬盘中数据文件的时间、不同节点间的数据传输时间及一部分处理器的处理时间等。各种因素决定了分布式文件系统的用户体验。即分布式文件系统中数据的读取速率不能与本地文件系统中数据的读取速率相差太大，否则在本地文件系统中打开一个文件需要 2 秒，而在分布式文件系统中在各种因素的影响下用时超过 10 秒，就会严重影响用户的使用体验。

（2）数据的存储方式。例如有 1000 万个数据文件，可以在一个节点存储全部数据文件，在其他 N 个节点每个节点存储 1000/N 万个数据文件作为备份；或者平均分配到 N 个节点存储，每个节点存储 1000/N 万个数据文件。无论采取何种存储方式，目的都是为了保证数据的存储安全和方便获取。

（3）数据的安全机制。由于数据分散在各个节点中，因而必须采取冗余、备份、镜像等方式保证在节点出现故障的情况下能够进行数据恢复，确保数据安全。

3.2 HDFS 文件系统

HDFS 是 Hadoop 项目的核心子项目，是分布式计算中数据存储管理的基础，是基于流数据模式访问和处理超大文件的需求而开发的，可以运行于廉价的商用服务器上。它所具有的高容错、高可靠性、高可扩展性、高获得性、高吞吐率等特征为海量数据提供了安全可靠的存储，为超大数据集（Large Data Set）的应用处理带来了很多便利。

3.2.1 HDFS 设计与架构

1. HDFS的概念和系统架构

在 HDFS 中，由 4 个基本要素构成了 HDFS 的系统体系，分别为元数据节点（NameNode）、数据节点（DataNode）、从元数据节点（Secondary NameNode）和客户端（Client）。四者之间的关系如图 3-1 所示。

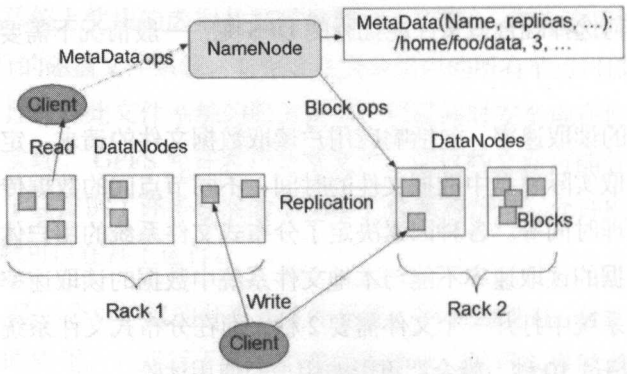


图 3-1　HDFS 系统构成和相互关系

我们来看看这 4 个要素的主要责任分别是什么。

1）元数据节点

可以将元数据节点看成分布式文件系统的管理者，只需用来管理文件系统的命名空间、存储块的备份和集群配置信息等。元数据节点会将文件系统的元数据（MetaData）存储在内存中，这些内容记录文件的基本信息，例如，此文件包括哪些数据块、这些数据块都分布在哪些数据节点上等。一个集群只有一个元数据节点的设计大大简化了系统架构。

2）数据节点

顾名思义，真正存储数据的地方，以数据块为单位，并周期性地将所有存储的数据块信息发送给元数据节点。客户端在和元数据节点沟通之后，可以向数据节点请求读出或者写入操作。

关于数据块，这里要特别说明的是，在默认情况下，HDFS 存储数据块的大小是 64MB，从这点上看和普通的文件系统相似。但不同于普通文件系统的是，

在 HDFS 中，如果一个文件小于一个数据块的大小，那么它并不占用整个数据块存储空间，而是有多少占用多少。

3）从元数据节点

所谓“从”一般有顺从、辅助的意思。同样，这里的从元数据节点并不是元数据节点的备用节点，而是与元数据配合工作，与元数据有不同的责任。

那么，为什么需要“从元数据节点”呢？在元数据中保存有命名空间镜像文件和修改日志（Edit Log）。例如，当文件系统客户端（Client）进行写操作时，首先把它记录在修改日志中，每次的写操作成功之前，修改日志都会同步（sync）到文件系统。但是当数据量和数据操作非常大的时候，就会出现日志文件过大的问题。为了防止日志文件过大，“从元数据节点”周期性地将元数据节点的命名空间镜像文件和修改日志合并，帮助元数据节点将内存中的元数据信息存储到硬盘上的。值得说明的是，其检查点在硬盘上的命名空间镜像文件是一种序列化的格式，并不能在硬盘上直接修改。合并后的命名空间镜像文件也在从元数据节点保存了一份，在元数据节点失败的时候可以恢复。

4）客户端

客户端（Client）就是需要获取 HDFS 系统中文件的应用程序和接口，引起 HDFS 的读 / 写等操作。

典型情况下，在 HDFS 集群中，有一个专门的机器运行元数据节点，集群中的其他机器各自运行一个数据节点。也允许将元数据节点和数据节点部署在同一机器上，甚至可以在物理上将多个数据节点部署在同一机器上。

2. HDFS 的数据流设计

我们了解了 HDFS 系统中的四大元素，那么这四大元素是如何相互配合工作的？也就是说 HDFS 是如何工作的呢？以下通过文件读取过程和文件写入过程进行详细介绍。

1）文件读取过程

简要地讲，文件读取过程为：客户端向元数据节点发起文件读取请求，元数据节点返回文件存储的数据节点的信息，客户端读取文件信息，如图 3-2 所示。

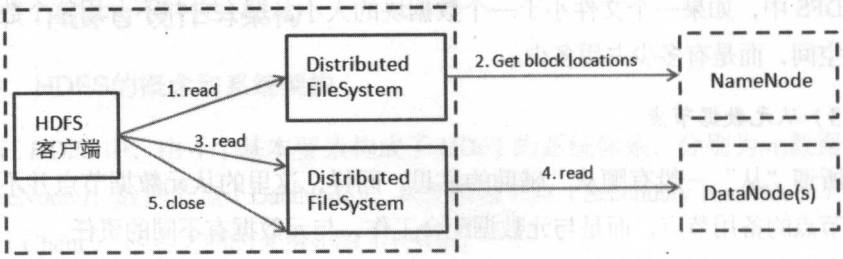


图 3-2　HDFS 文件读取过程

详细过程如下：

（1）客户端（Client）调用 FileSystem 的 open() 函数打开文件。

（2）DistributedFileSystem 用 RPC 调用元数据节点，得到文件的数据块信息。

（3）对于每一个数据块，元数据节点返回保存数据块的数据节点的地址。

（4）DistributedFileSystem 返回 FSDataInputStream 给客户端，用来读取数据。

（5）客户端调用 stream 的 read() 函数开始读取数据。

（6）DFSInputStream 连接保存此文件第一个数据块的最近的数据节点。

（7）Data 从数据节点读到客户端（Client）。

（8）当此数据块读取完毕后，DFSInputStream 关闭和此数据节点的连接，然后连接此文件下一个数据块的最近的数据节点。

（9）当客户端读取数据完毕后，调用 FSDataInputStream 的 close() 函数。

（10）在读取数据的过程中，如果客户端在与数据节点通信时出现错误，则尝试连接包含此数据块的下一个数据节点。

（11）失败的数据节点将被记录，以后不再连接。

2）文件写入过程

简要地讲，文件写入过程为：客户端向元数据节点发起文件写入请求，元数据节点根据文件大小和文件块配置情况，返回给客户端它所管理部分数据节点的信息；客户端将文件划分为多个数据块，根据数据节点的地址信息，按顺序写入到每一个数据块中，如图 3-3 所示。

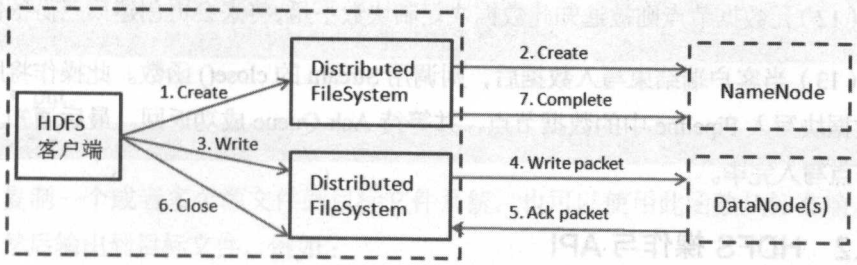


图 3-3 HDFS 文件写入过程

详细过程如下：

（1）客户端调用 create() 函数创建文件。

（2）DistributedFileSystem 用 RPC 调用元数据节点，在文件系统的命名空间中创建一个新的文件。

（3）元数据节点首先确定文件原来不存在，并且客户端有创建文件的权限，然后创建新文件。

（4）DistributedFileSystem 返回 DFSOutputStream，客户端用于写数据。

（5）客户端开始写入数据，DFSOutputStream 将数据分成块，写入 Data Queue。

（6）Data Queue 由 Data Streamer 读取，并通知元数据节点分配数据节点，用来存储数据块（每块默认复制 3 块）。分配的数据节点放在一个 Pipeline 里。

（7）Data Streamer 将数据块写入 Pipeline 中的第一个数据节点；第一个数据节点将数据块发送给第二个数据节点；第二个数据节点将数据块发送给第三个数据节点。

（8）DFSOutputStream 为发出去的数据块保存了 Ack Queue，等待 Pipeline 中的数据节点告知数据已经成功写入。

（9）如果数据节点在写入的过程中失败，则关闭 Pipeline，将 Ack Queue 中的数据块放入 Data Queue 的开始。

（10）当前的数据块在已经写入的数据节点中被元数据节点赋予新的标识，则错误节点重启后能够察觉其数据块是过时的，会被删除。

（11）失败的数据节点从 Pipeline 中移除，另外的数据块则写入 Pipeline 中的另外两个数据节点。

（12）元数据节点则被通知此数据块复制块数不足，将来会再创建第三份备份。

（13）当客户端结束写入数据后，则调用 Stream 的 close() 函数。此操作将所有的数据块写入 Pipeline 中的数据节点，并等待 Ack Queue 成功返回。最后通知元数据节点写入完毕。

3.2.2 HDFS 操作与 API

Apache HDFS 提供了文件系统 FS Shell 交互式命令操作，通过这些操作可以直接操作 HDFS。基本命令格式如下：

```
bin/hadoop fs <args>
```

所有的 FS Shell 命令都以 URL 作为参数（<args>）。

这里 URL 的格式为：

```
scheme://authority/path
```

对于 HDFS，scheme 为 hdsf；对于本地文件系统，scheme 为 file。scheme 和 authority 是可选的。如果没有指定，则将使用配置文件中的 scheme 和 authority。

一个 HDFS 文件或者文件夹，比如 /parent/child 可以指定为 hdfs://namenodehost/parent/child 或者 /parent/child（在此种情况下，将指向配置中的 hdfs://namenodehost）。举例来说，可以指定为 hdfs://localhost:9000/user/tester/test.txt，假设已经在 core-site.xml 里配置了 fs.default.name=hdfs://localhost:9000，则仅使用 /user/tester/test.txt 即可。

其实我们在使用 HDFS FS Shell 操作命令的时候，可以类似于使用普通的 Shell，如果执行过程中出错，将会弹出相应的错误信息，执行结束后，返回值的写法也和 Shell 类似。错误流将输出到 stderr，其他将输出到 stdout。如果在程序中调用 FS Shell 操作，则可以直接通过这种方式监控。

以下列举最常见的 HDFS FS Shell 操作，详细信息建议参考 Apache HDFS 官网。

- ls。

```
hadoop fs -ls [-d] [-h] [-R] [-t] [-S] [-r] [-u] <args>
```

列出 HDFS 文件系统根目录下的目录和文件，例如：

```
hadoop fs -ls /user/hadoop/file
hadoop fs -lsr /user //递归列出目录下的所有文件
```

- put。

```
hadoop fs -put <localsrc> ... <dst>
```

复制一个或者多个源文件到目标文件系统。也可以使用此函数从标准输入读取，然后输出到目标文件，例如：

```
hadoop fs -put localfile /user/hadoop/hadoopfile
hadoop fs -put localfile1 localfile2 /user/hadoop/hadoopdir
hadoop fs -put localfile hdfs://nn.example.com/hadoop/hadoopfile
hadoop fs -put - hdfs://nn.example.com/hadoop/hadoopfile //从标准输入中读取
```

还有两个操作的功能和 put 类似：moveFromLocal 和 copyFromLocal。

- get。

```
hadoop fs -get [-ignorecrc] [-crc] <src> <localdst>>
```

local file 和 hdfs file 的名字不能相同，否则会提示文件已存在，没有重名的文件会复制到本地。

```
hadoop fs -get < hdfs file or dir > ... < local dir >
```

复制多个文件或目录到本地时，本地要为文件夹路径。例如：

```
hadoop fs -get /user/hadoop/file localfile
hadoop fs -get hdfs://nn.example.com/user/hadoop/file localfile
```

注意：如果用户不是 root，则本地路径必须是用户文件夹下的路径，否则会出现权限问题。

还有两个函数的功能与 get 类似：copyToLocal 和 moveToLocal。

- rm。

```
hadoop fs -rm [-f] [-r |-R] [-skipTrash] URI [URI ...]
```

删除文件或者目录，-R 可以递归删除文件夹下的文件。例如：

```
hadoop fs -rm hdfs://nn.example.com/file /user/hadoop/emptydir
```

- mkdir。

```
hadoop fs -mkdir < hdfs path>
```

只能一级一级地创建目录，如果父目录不存在，则使用这个命令会报错。

```
hadoop fs -mkdir -p < hdfs path>
```

如果所创建目录的父目录不存在，就创建该父目录。例如：

```
hadoop fs -mkdir /user/hadoop/dir1 /user/hadoop/dir2
hadoop fs -mkdir hdfs://nn1.example.com/user/hadoop/dir hdfs://nn2.
example.com/user/hadoop/dir
```

- getmerge。

```
hadoop fs -getmerge <src> <localdst> [addnl]
```

将 HDFS 指定目录下的所有文件排序后合并到 Local 指定的文件中，文件不存在时会自动创建，文件存在时会覆盖里面的内容。

```
hadoop fs -getmerge -nl < hdfs dir > < local file >
```

加上 -nl 后，合并到 Local File 中的 HDFS 文件之间会空出一行。

- cp。

```
hadoop fs -cp [-f] [-p | -p[topax]] URI [URI ...] <dest>
```

复制。目标文件不能存在，否则命令不能执行。相当于给文件重命名并保存，源文件还存在。-f 将会直接覆盖文件夹。

```
hadoop fs -cp < hdfs file or dir >... < hdfs dir >
```

目标文件夹必须存在，否则命令不能执行。例如：

```
hadoop fs -cp /user/hadoop/file1 /user/hadoop/file2
hadoop fs -cp /user/hadoop/file1 /user/hadoop/file2 /user/hadoop/dir
```

- mv。

```
hadoop fs -mv < hdfs file > < hdfs file >
```

目标文件不能存在，否则命令不能执行。相当于给文件重命名并保存，源文件不存在。

```
hadoop fs -mv < hdfs file or dir >... < hdfs dir >
```

源路径有多个时，目标路径必须为目录，且必须存在。值得注意的是，跨文件系统的移动（Local 到 HDFS 或者反过来）是不允许的。例如：

```
hadoop fs -mv /user/hadoop/file1 /user/hadoop/file2
hadoop fs -mv hdfs://nn.example.com/file1 hdfs://nn.example.com/file2
hdfs://nn.example.com/file3 hdfs://nn.example.com/dir1
```

- count。

```
hadoop fs -count < hdfs path >
```

统计 HDFS 对应路径下的目录个数、文件个数、文件总计大小，显示为目录个数、文件个数、文件总计大小、输入路径。例如：

```
hadoop fs -count hdfs://nn1.example.com/file1 hdfs://nn2.example.com/file2
hadoop fs -count -q hdfs://nn1.example.com/file1
hadoop fs -count -q -h hdfs://nn1.example.com/file1
hdfs dfs -count -q -h -v hdfs://nn1.example.com/file1
```

- du。

```
hadoop fs -du < hdsf path>
```

显示 HDFS 对应路径下每个文件夹和文件的大小。

```
hadoop fs -du -s < hdsf path>
```

显示 HDFS 对应路径下所有文件的大小总和。

```
hadoop fs -du -h < hdsf path>
```

显示 HDFS 对应路径下每个文件夹和文件的大小。文件的大小用方便阅读的形式表示，例如用 64MB 代替 67108864。例如：

```
hadoop fs -du /user/hadoop/dir1 /user/hadoop/file1 hdfs://nn.example.
com/user/hadoop/dir1
```

- text。

```
hadoop fs -text < hdsf file>
```

将文本文件或某些格式的非文本文件通过文本格式输出。

- setrep。

```
hadoop fs -setrep -R 3 < hdfs path >
```

改变一个文件在 HDFS 中的副本个数。上述命令中数字 3 为设置的副本个数，-R 选项可以对一个目录下的所有目录 + 文件递归执行改变副本个数的操作。例如：

```
hadoop fs -setrep -w 3 /user/hadoop/dir
```

- stat。

```
hdoop fs -stat [format] < hdfs path >
```

- tail。

```
hadoop fs -tail < hdfs file >
```

在标准输出中显示文件末尾的 1KB 数据。例如：

```
hadoop fs -tail pathname
```

3.2.3 HDFS 的优点及适用场景

从 HDFS 的设计理念和实际使用效果来看，HDFS 的优点和适用场景如下。

1. 存储超大的数据

在大数据应用场景中，一般数据量都非常大，起步就是 GB 级别，甚至达到 TB 级别。HDFS 适合应用于这些数据量大的场景下，其性能才能得到有效的发挥。目前已经有案例将 HDFS 应用到 PB 级别的环境中。也有案例将 HDFS 结合 MapReduce 部署到万台以上的集群中，处理百万规模的文件数量。

2. 离线的数据

在 HDFS 设计理念中，有一条为“一次写入，多次读写”。什么意思呢？它是指每个磁盘或机器从数据源复制数据到本地，即一次写入，然后长时间地只需分析时读取本地数据，即多次读取。这意味着一个数据集一旦由数据源生成，就会被复制分发到不同的存储节点中，然后响应各种各样的数据分析任务请求。在数据分析过程中，可能会涉及多个数据块，那么 HDFS 请求读取多个甚至全部数据集的速度要快很多。

3. 容错性强

HDFS 具有较强的容错性。HDFS 采用 Replication 策略实现较强的容错性。当某个数据块不可访问时，HDFS 将自动复制相应的块从而维持备份的次数。

4. 采用 Block 机制

HDFS 采用 Block，使得文件可以非常大，因为 Block 可以存储在集群上的任意一台机器上。这样就具有了分布式文件系统的基础，比如可以做负载平衡、可以做冗余以避免文件损坏或机器故障等。

5. 结构简化高效

从结构上讲，采用从数据节点和数据节点的设计，责任分工明确。一个集群只有一个元数据节点的设计大大简化了系统架构。存储子系统的职责单一化。只负责 Block 的读 / 写，文件的权限控制等可由其他系统负责。

6. 运行于廉价的商用机器集群上

HDFS 设计对硬件需求比较低，只需运行在低廉的商用硬件集群上，而无须运行在昂贵的高可用性机器上。廉价的商用机也就意味着大型集群中出现节点故障的概率非常高。这就要求设计 HDFS 时要充分考虑数据的可靠性、安全性及高可用性。

3.2.4 HDFS 的缺点及改进策略

1. 不适合低延迟数据访问

如果要处理一些用户要求时间比较短的低延迟应用请求，则 HDFS 不适合。HDFS 是为了处理大型数据集分析任务的，主要是为达到高的数据吞吐量而设计的，这就可能要求以高延迟作为代价。

改进策略：对于那些有低延迟要求的应用程序，HBase 是一个更好的选择。HBase 通过上层数据管理项目来尽可能地弥补这个不足，在性能上有了很大的提升。使用缓存或多 Master 设计可以降低 Client 的数据请求压力，以减少延迟。还有就是对 HDFS 系统内部的修改，这就需要权衡大吞吐量与低延迟，HDFS 不是万能的。

2. 无法高效存储大量小文件

因为 NameNode 把文件系统的元数据放置在内存中，所以文件系统所能容纳的文件数目是由 NameNode 的内存大小决定的。一般来说，每一个文件、文件夹和 Block 需要占据 150 字节左右的空间，所以，如果你有 100 万个文件，每个文件占据一个 Block，那么你至少需要 300MB 内存。就当前来说，数百万的文件还是可行的，但当扩展到数十亿水平时，对于当前的硬件水平来说就无法实现了。还有一个问题是，因为 MapTask 的数量是由 splits 来决定的，所以用 MapReduce 处理大量的小文件时，就会产生过多的 MapTasks，线程管理开销

将会增加。举个例子，处理 10 000MB 的文件，若每个 split 为 1MB，那就会有 10 000 个 MapTasks，会有很大的线程开销；若每个 split 为 100MB，则只有 100 个 MapTasks，每个 MapTask 将会有更多的事情做，而线程的管理开销也将减少很多。

改进策略：要想让 HDFS 处理好小文件，有不少方法可以选择。

- 利用 SequenceFile、MapFile、Har 等方式归档小文件。这种方法的原理就是把小文件归档起来管理，HBase 就是基于此原理的。对于这种方法，如果想找回原来的小文件内容，就必须知道与归档文件的映射关系。
- 横向扩展。一个 Hadoop 集群能管理的小文件有限，那就把几个 Hadoop 集群拖到一台虚拟服务器后面，形成一个大的 Hadoop 集群。Google 就这么做过。
- 多 Master 设计。这种方法的作用显而易见。正在研发中的 GFS II 也要改为分布式多 Master 设计，还支持 Master 的 Failover，而且 Block 的大小也改为 1MB，有意调优处理小文件。

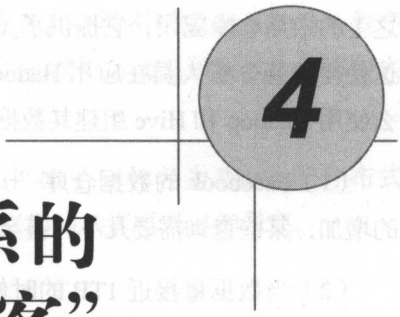
3. 不支持多用户写入及任意修改文件

在 HDFS 的一个文件中只有一个写入者，而且写操作只能在文件末尾完成，即只能执行追加操作。目前 HDFS 还不支持多个用户对同一文件的写操作，以及在文件任意位置进行修改。

3.3 小结

本章分析了各种 DFS，并对 HDFS 进行了详细阐述。HDFS 被设计成适合运行在通用硬件上的分布式文件系统，它和现有的分布式文件系统有很多共同点。坦白说 HDFS 是一个不错的分布式文件系统，它有很多优点，但也有一些缺点，包括：不适合低延迟数据访问、无法高效存储大量小文件、不支持多用户写入及任意修改文件。HDFS 在大数据处理领域占据了很大的席位，HDFS 是 Apache Hadoop Core 很重要的一部分。

除了 HDFS，还有其他的文件系统，比如 IBM GPFS，在大规模的大数据处理集群中是一个很好的选择。



Hadoop 体系的“四剑客”

4.1 数据仓库工具 Hive

4.1.1 Hive 缘起何处

Hive 是什么东西？翻译一下是“蜂巢”的意思。相信有不少读者和笔者一样，小时候喜欢玩的一种游戏是用弹弓打蜂巢，然后被蜜蜂追着跑。躲过了蜜蜂的追逐，就有后福了——吃打下来的巢脾里面的蜜，堪称“舌尖上的中国”。那么重点来了，笔者小时候吃蜂蜜的时候就在想一个问题：上千只蜜蜂同时建造一个蜂巢，没有图纸，没有设计，没有人监工，小蜜蜂能建造出结构相当精良的 Hive，是多么不可思议的一件事情！笔者相信 Hive 的创始者也一定认识到了这一点，才会将 Hadoop 这么关键的技术命名为 Hive，其用意可能是将 Hive 建成结构精良的数据仓库。

那么，什么是 Hive 呢？ Hive 是一种数据仓库工具。首先要明确的是，Hive 只是一种工具。那么这种工具是用来干什么的呢？可以用 Hive 将结构化的数据文件映射为一张数据库表，并提供完整的 SQL 查询功能，可以将 SQL 语句转换为 MapReduce 任务运行。所以 Hive 本身并不具备存储功能，要将它和数据库区分开，后面我们还会详细区分。

那么 Hive 是怎么来的呢？事实上，只要我们搞清楚了 Hive 的来源和初衷，

就能深刻地理解 Hive 是干什么的。Hive 是 Facebook 开发的构建于 Hadoop 集群之上的数据仓库应用，它提供了类似于 SQL 语法的 HQL 语句作为数据访问接口，这使得普通分析人员在应用 Hadoop 的过程中更加容易学习。至于 Facebook 为什么使用 Hadoop 和 Hive 组建其数据仓库，大致的原因如下：

（1）Facebook 的数据仓库一开始是构建于 MySQL 之上的，但是随着数据量的增加，某些查询需要几小时甚至几天才能完成。

（2）当数据量接近 1TB 的时候，MySQL 后台进程宕掉，这时 Facebook 决定将数据仓库转移到 Oracle。当然这次转移过程也是付出了很大代价的，比如支持的 SQL 方言不同、修改以前的运行脚本等。

（3）Oracle 应付几 TB 的数据还是没有问题的，但是在开始收集用户点击流的数据（每天大约 400GB）之后，Oracle 也开始撑不住了，由此又要考虑新的数据仓库方案。

（4）内部开发人员花了几周的时间建立了一个并行日志处理系统 Cheetah，这样勉强可以在 24 小时内处理完一天的点击流数据。

（5）Cheetah 也存在许多缺点。后来偶然发现了 Hadoop 项目，并开始试着将日志数据同时载入 Cheetah 和 Hadoop 进行对比，发现 Hadoop 在处理大规模数据时更具优势。后来将所有的工作流都从 Cheetah 转移到 Hadoop，并基于 Hadoop 做了很多有价值的分析。

（6）后来为了使大多数人能够使用 Hadoop，便开发了 Hive。Hive 提供了类似于 SQL 的查询接口，非常方便。与此同时还开发了一些其他工具。

（7）现在集群存储 2.5PB 的数据，并且以每天 15TB 的数据在增长，每天提交 3000 个以上的作业，大约处理 55TB 的数据。

总体上看，Hive 有两个典型的特点：其一，Hive 是建立在 Hadoop 上的数据仓库基础架构；其二，较低的学习代价便可以让用户在 Hadoop 中存储、查询和分析大规模数据，如图 4-1 所示。

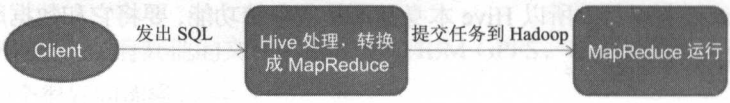


图 4-1　Hive 的作用

简单地理解：如果你只需完成大规模数据分析这件事情，那么，你只需有一套 Hadoop 环境加上一个 Hive 数据库，只要你懂 SQL，而不必懂 MapReduce 程序如何编程及 Hadoop 底层如何工作，你的 SQL 需求将被自动编译到整个集群中去进行分布式计算，以提高分析效率。

顺便提一句，有关资料显示，Facebook 又推出了一种名为 Presto 的分布式 SQL 查询引擎，据说性能要比 Hive 强很多，感兴趣的读者可以详细研究。

4.1.2 Hive 和数据库的区别

由于 Hive 采用了 SQL 的查询语言 HQL，有些读者很容易将 Hive 理解为数据库。其实从专业角度来说，Hive 并不是数据库。从结构上来看，Hive 和数据库除了拥有类似的查询语言，再无类似之处。数据库可以用在 Online 的应用中，但是 Hive 是为数据仓库而设计的。清楚这一点，有助于从应用角度理解 Hive 的特性。

（1）数据存储位置。Hive 是建立在 Hadoop 之上的，所有 Hive 的数据都是存储在 HDFS 中的；而数据库则可以将数据保存在块设备或者本地文件系统中。

（2）查询语言。由于 SQL 被广泛地应用在数据仓库中，因此专门针对 Hive 的特性设计了类 SQL 的查询语言 HQL。熟悉 SQL 开发的开发者可以很方便地使用 Hive 进行开发。

（3）索引。Hive 在加载数据的过程中不会对数据进行任何处理，甚至不会对数据进行扫描，因此也没有对数据中的某些 key 建立索引。Hive 要访问数据中满足条件的特定值时，需要暴力扫描整个数据，因此访问延迟较高。由于 MapReduce 的引入，Hive 可以并行访问数据，因此即使没有索引，对于大数据量的访问，Hive 仍然可以体现出优势。值得一提的是，Hive 在 0.8 版本之后引入了图索引。在数据库中，通常会针对一列或者几列建立索引，因此对于少量的、特定条件的数据的访问，数据库可以有很高的效率和较低的延迟。由于数据的访问延迟较高，决定了 Hive 不适合在线数据查询。

（4）数据格式。Hive 中没有定义专门的数据格式，数据格式可以由用户指定。用户定义数据格式需要指定三个属性：列分隔符（通常为空格、“\t”、“\x001”）、行分隔符（\n）及读取文件数据的方法（Hive 中默认有三种文件格式：TextFile、SequenceFile 及 RCFile）。由于在加载数据的过程中不需要从用户定义的数据格式

到 Hive 定义的数据格式的转换，因此，Hive 在加载过程中不会对数据本身进行任何修改，而只是将数据内容复制或者移动到相应的 HDFS 目录中。而在数据库中，不同的数据库有不同的存储引擎，而且定义了自己的数据格式。所有数据都会按照一定的组织存储，因此，数据库加载数据的过程会比较耗时。

（5）执行。Hive 中大多数查询的执行是通过 Hadoop 提供的 MapReduce 来实现的（类似于 select * from tbl 的查询不需要 MapReduce），而数据库通常有自己的执行引擎。

（6）数据更新。由于 Hive 是针对数据仓库应用设计的，而数据仓库的内容是读多写少的，因此，Hive 中不支持对数据的改写和添加，所有数据都是在加载的时候确定好的。而数据库中的数据通常是需要修改的，因此可以使用 INSERT INTO ... VALUES 添加数据，使用 UPDATE ... SET 修改数据。

（7）延迟。之前提到，Hive 在查询数据的时候由于没有索引，需要扫描整张表，因此延迟较高。另外一个导致 Hive 执行延迟高的因素是 MapReduce 框架。由于 MapReduce 本身具有较高的延迟，因此在利用 MapReduce 执行 Hive 查询时，也会有较高的延迟。相对地，数据库的执行延迟较低。当然，这个低是有条件的，即数据规模较小。当数据规模大到超过数据库的处理能力的时候，Hive 的并行计算显然能体现出优势。

（8）可扩展性。由于 Hive 是建立在 Hadoop 之上的，因此 Hive 的可扩展性和 Hadoop 的可扩展性是一致的；而数据库由于 ACID 语义的严格限制，扩展性非常有限。目前最先进的并行数据库 Oracle 在理论上的扩展能力也只有 100 台左右。

（9）数据规模。由于 Hive 建立在集群上并且可以利用 MapReduce 进行并行计算，因此可以支持很大规模的数据；而数据库可以支持的数据规模较小，一般最大为 20 个节点左右。

（10）硬件要求。Hive 对硬件的要求相对较低；而关系型数据库管理系统为了提高线上处理速度，对硬件的要求相对较高。

4.1.3 Hive 设计思想与架构

1. Hive 体系的组成

Hive 的体系结构如图 4-2 所示。

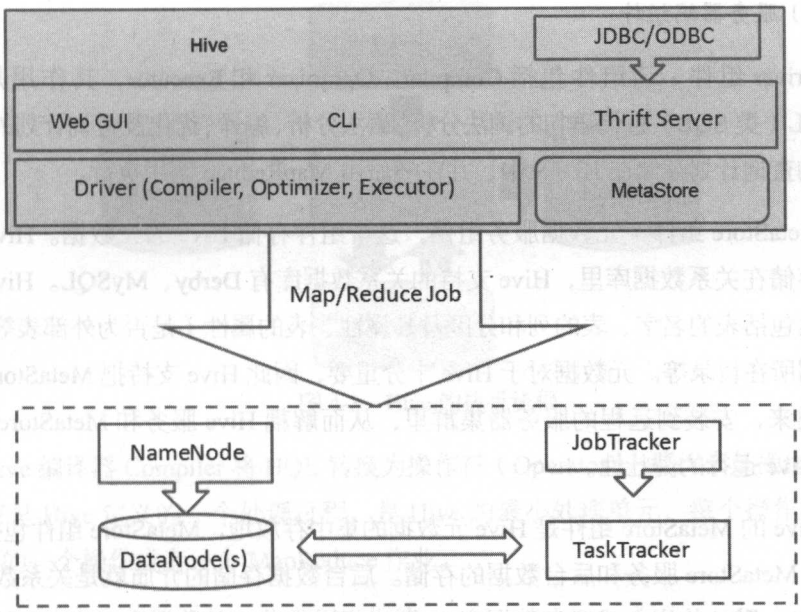


图 4-2 Hive 的体系结构

由图 4-2 可以看出，Hive 是建立在 Hadoop 基础上的，是针对 Hadoop MapReduce 开发的技术。Hive 的组件包括 CLI（Command Line Interface）、JDBC/ODBC、Thrift Server、Web GUI、MetaStore 和 Driver（Compiler、Optimizer 和 Executor）。其实这么多组件大致可以分为两类：客户端组件和服务端组件。下面我们对这些组件进行逐一说明。

1）客户端组件

CLI：Command Line Interface，命令行接口。最常用的客户端组件就是 CLI，CLI 启动的时候，会同时启动一个 Hive 副本。

Client（客户端）：Client 是 Hive 的客户端，用于连接 Hive Server。在启动 Client 模式的时候，需要指出 Hive Server 所在的节点，并且在该节点启动 Hive Server。图 4-2 所示的架构图里没有写上 Thrift 客户端，但是 Hive 架构的许多客户端接口都是建立在 Thrift 客户端之上的，包括 JDBC 和 ODBC 接口。

Web GUI：Hive 客户端提供了一种通过网页访问 Hive 所提供的服务的方式。这个接口对应 Hive 的 HWI（Hive Web Interface）组件，使用前要启动 HWI 服务。

2）服务器端组件

Driver 组件：该组件包括 Compiler、Optimizer 和 Executor，其作用是完成 HiveQL（类 SQL）查询语句的词法分析、语法分析、编译、优化及查询计划的生成。生成的查询计划存储在 HDFS 中，并在随后由 MapReduce 调用执行。

MetaStore 组件：元数据服务组件，这个组件存储 Hive 的元数据。Hive 的元数据存储在关系数据库里，Hive 支持的关系数据库有 Derby、MySQL。Hive 中的元数据包括表的名字、表的列和分区及其属性、表的属性（是否为外部表等）、表的数据所在目录等。元数据对于 Hive 十分重要，因此 Hive 支持把 MetaStore 服务独立出来，安装到远程的服务器集群里，从而解耦 Hive 服务和 MetaStore 服务，保证 Hive 运行的健壮性。

Hive 的 MetaStore 组件是 Hive 元数据的集中存放地。MetaStore 组件包括两个部分：MetaStore 服务和后台数据的存储。后台数据存储的介质就是关系数据库，例如 Hive 默认的嵌入式磁盘数据库 Derby，还有 MySQL 数据库。MetaStore 服务是建立在后台数据存储介质之上，并且可以和 Hive 服务进行交互的服务组件。默认情况下，MetaStore 服务和 Hive 服务是安装在一起的，运行在同一个进程当中。也可以把 MetaStore 服务从 Hive 服务中剥离出来，独立安装在一个集群里，Hive 远程调用 MetaStore 服务。我们可以把元数据这一层放到防火墙之后，当客户端访问 Hive 服务时，就可以连接到元数据这一层，从而提供更好的管理性能和安全保障。使用远程的 MetaStore 服务，可以让 MetaStore 服务和 Hive 服务运行在不同的进程里，这样既保证了 Hive 的稳定性，又提升了 Hive 服务的效率。

Thrift 服务：Thrift 是 Facebook 开发的一个软件框架，它用来进行可扩展且跨语言服务的开发。Hive 集成了该服务，可以让不同的编程语言调用 Hive 的接口。

2. Hive 的执行流程

Hive 的执行流程是将 HQL 语句转化为一系列可以在 Hadoop MapReduce 集群中运行的 Job。通常在客户端执行 Hive 命令，然后输入 SQL 语句，Hive 将 SQL 语句生成多个 MapReduce 的 Job，然后将这些 Job 提交给 Hadoop 执行，执行完成后，再把结果放入 HDFS 或者本地的临时文件中。

Hive 大致的执行流程如图 4-3 所示。

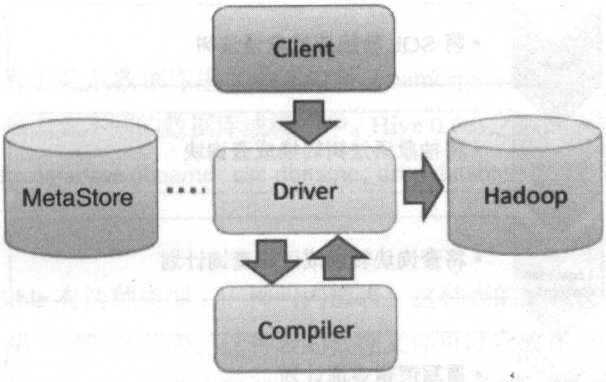


图 4-3 Hive 的执行流程

Hive 编译器 Compiler 将 HQL 转换为操作符（Operator）。那什么是操作符呢？操作符是 Hive 定义的一个处理过程，是 Hive 的最小处理单元，每个操作符代表 HDFS 的一个操作或者一个 MapReduce 作业。

以下列举出 Hive 中的一些常用操作符。

- TableScanOperator：扫描 Hive 表数据。
- RinkOperator：创建将发送到 Reducer 端的 <key,value> 对。
- JoinOperator：连接两份数据。
- SelectOperator：选择输出列。
- FileSinkOperator：建立结果数据，输出至文件。
- FilterOperator：过滤输入数据。
- GroupByOperator：GroupBy 语句。
- MapJoinOperator：/*+mapjoin(t) */。
- LimitOperator：Limit 语句。
- UnionOperator：Union 语句。

Hive 的数据存储在 HDFS 中，大部分的查询由 MapReduce 完成（包含 * 的查询，比如 select * from tbl 不会生成 MapRedcue 任务）。Hive 通过 ExecMapper 和 ExecReducer 执行 MapReduce 任务。在执行 MapReduce 任务时有两种模式，即本地模式和分布式模式。

Hive 编译器的组成如图 4-4 所示。

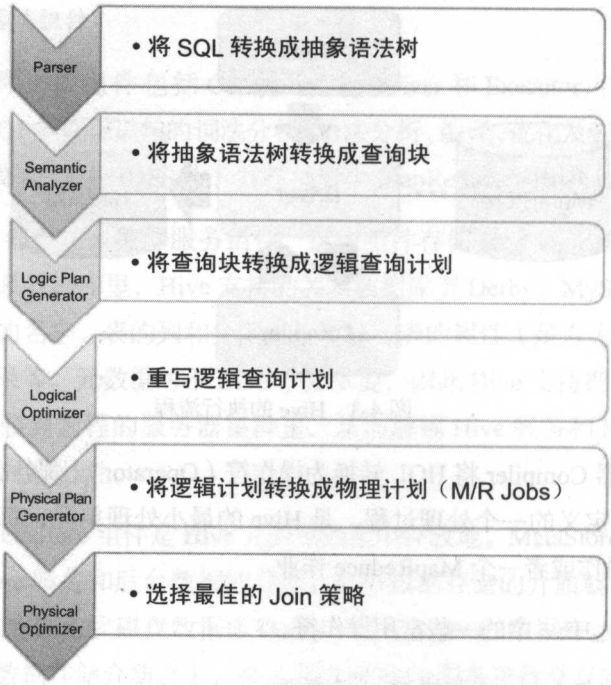


图 4-4　Hive 编译器的组成

Hive 对一条 SQL 语句的执行过程大致为：一条 SQL 语句（以查询 SQL 为例）的最终目的是将一张表或者若干张表中的所有行数据一条一条地进行处理，最终生成一组目标记录。为了实现这样的目的，首先将处理过程分解为若干个算子，然后将初始的表数据记录依次通过这些算子进行计算，最终得出结果。例如，select a from tbl where b>1 order by c，对于这条 SQL 语句，首先需要一个 ts（table scan）算子从表中读出数据，然后读出的数据经过一个 fil（filter）算子过滤那些不满足条件 b>1 的数据，最后经过一个 fetch 算子将正确的数据返回。对于任意复杂的 SQL 语句，都可以生成这样的算子树进行处理。

3. Hive 数据模型与存储

首先，Hive 没有专门的数据存储格式，也没有为数据建立索引，用户可以非常自由地组织 Hive 中的表，只需在创建表的时候告诉 Hive 数据中的列分隔符和行分隔符，Hive 就可以解析数据。其次，Hive 中的所有数据都存储在 HDFS 中。Hive 中包含 4 种数据模型：Database、Table、Partition 和 Bucket。下面一一论述这 4 种数据模型。

1）Database

Database 相当于关系数据库里的命名空间（namespace），它的作用是将用户和数据库的应用隔离到不同的数据库或模式中。Hive 0.6.0 之后的版本支持该模型。Hive 提供了 create database dbname、use dbname、drop database dbname 这样的语句。

2）Table

Hive 里的 Table 有两种类型：一种叫内部表，这种表的数据文件存储在 Hive 的数据仓库里；另一种叫外部表，这种表的数据文件可以存放在 Hive 数据仓库外部的分布式文件系统上，也可以放到 Hive 数据仓库里（注意：Hive 的数据仓库也就是 HDFS 上的一个目录，这个目录是 Hive 数据文件存储的默认路径，它可以在 Hive 的配置文件里进行配置，最终也会存放到元数据库里）。

Hive 的内部表和数据库中的 Table 在概念上是类似的，每张 Table 在 Hive 中都有一个相应的目录存储数据。Table 逻辑上由存储的数据和描述表格中的数据形式的相关元数据组成。表存储的数据存放在分布式文件系统里，例如 HDFS，元数据存储在关系数据库里。当我们创建一张 Hive 的表，还没有为表加载数据的时候，该表在分布式文件系统（例如 HDFS）上就是一个文件夹（文件目录）。例如一张表 pvs，它在 HDFS 中的路径为 /where/pvs，其中，where 是在 hive-site.xml 中由 ${hive.metastore.warehouse.dir} 指定的数据仓库的目录，所有的 Table 数据（不包括 External Table）都保存在这个目录中。

下面是关于内部表的操作示例。

创建内部表：

```
create table test_inner_table (key string)
```

加载数据：

```
LOAD DATA LOCAL INPATH 'filepath' INTO TABLE tenner_table
```

查看数据：

```
select * from test_inner_table;  select count(*) from tenner_table
```

删除表：

```
drop table test_inner_table
```

下面是关于外部表的操作示例。

创建外部表：

```
create external table test_external_table (key string)
```

加载数据：

```
LOAD DATA INPATH 'filepath' INTO TABLE test_inner_table
```

查看数据：

```
select * from test_external_table; select count(*) from test_external_table
```

删除表：

```
drop table test_external_table
```

在创建外部表时，在 table 之前要加关键字 external，同时还要用 location 命令指定文件存储的路径。如果不使用 locaction 命令，数据文件也会放置到 Hive 的数据仓库里。外部表指向已经在 HDFS 中存在的数据，可以创建 Partition。它和内部表在元数据的组织上是相同的，而实际数据的存储则有较大的差异。内部表的创建过程和数据加载过程可以独立完成，也可以在同一条语句中完成。在加载数据的过程中，实际数据会被移动到数据仓库目录中，之后对数据的访问将会直接在数据仓库目录中完成。删除表时，表中的数据和元数据将会被同时删除。而外部表只有一个过程，加载数据和创建表同时完成（CREATE EXTERNAL TABLE … LOCATION），实际数据存储在 LOCATION 后面指定的 HDFS 路径中，并不会移动到数据仓库目录中。

这两种表在使用上的区别主要是 drop 命令。drop 是 Hive 删除表的命令，内部表在执行 drop 命令的时候，会删除元数据和存储的数据；而外部表在执行 drop 命令的时候，只删除元数据库里的数据，而不会删除存储的数据。简单地说，删除内部表时，元数据与数据都会被删除；而删除外部表时，仅删除该链接。

3）Partition

Partition 是分区的意思。Hive 里 Partition 的概念是根据“分区列”的值对表的数据进行粗略划分的机制，在 Hive 存储上就体现在表的主目录（Hive 的表实际显示就是一个文件夹）下的一个子目录，这个文件夹的名字就是我们定义的分区列的名字。没有实际操作经验的人可能会认为分区列是表的某个字段，其实不是这样的，

分区列不是表的某个字段，而是独立的列，我们根据这个列来存储表中的数据文件。使用分区是为了加快数据的查询速度。我们在查询某个具体分区列里的数据时，没必要进行全表扫描。Partition 对应于数据库中的 Partition 列的密集索引，但是 Hive 中 Partition 的组织方式和数据库中的完全不同。在 Hive 中，表中的一个 Partition 对应于表下的一个目录，所有 Partition 的数据都存储在对应的目录中。

下面是关于 Partition 的简单操作示例。

创建表：

```
create table test_partition_table (key string) partitioned by (dt string)
```

加载数据：

```
LOAD DATA INPATH 'filepath' INTO TABLE test_partition_table partition
(dt='2006')
```

查看数据：

```
select * from test_partition_table;  select count(*) from test_
partition_table
```

删除表：

```
drop table test_partition_table
```

4）Bucket

Bucket 是水桶的意思，在大多数情况下，人们称之为“桶”。Table 和 Partition 都是目录级别的拆分数据，而 Bucket 则是针对数据源数据文件本身来拆分数据的。使用桶的表会将源数据文件按一定规律拆分成多个文件。要使用 Bucket，首先要打开 Hive 对桶的控制，命令如下：

```
set hive.enforce.bucketing = true
```

其实如果没有特别的场景或者特别的查询，则没有必要开启 Hive 对桶的控制。因为桶运用的场景有限，一个是做 Map 连接的运算，另外一个就是取样操作。

Bucket 是将表的列通过 Hash 算法进一步分解成不同的文件存储。它对指定列计算 Hash，根据 Hash 值切分数据，目的是为了并行，每个 Bucket 对应一个文件。例如，将 user 列分散至 32 个 Buckets，首先对 user 列的值计算 Hash，对应 Hash 值为 0 的 HDFS 目录为 /wh /pvs/ds=20090801/ctry=US/part-00000，对应 Hash 值为

20 的 HDFS 目录为 /wh/pvs/ds=20090801/ctry=US/part-00020。如果想应用更多的 Map 任务，这样做是不错的选择。

下面是关于 Bucket 的简单操作示例。

创建表：

```
create table test_bucket_table (key string) clustered by (key) into 20 buckets
```

加载数据：

```
LOAD DATA INPATH 'filepath' INTO TABLE test_bucket_table
```

查看数据：

```
select * from test_bucket_table
```

4.1.4 适用场景

Hive 构建在基于静态批处理的 Hadoop 之上，Hadoop 通常都有较高的延迟并且在作业提交和调度的时候需要大量的开销。Hive 常用于数据分析，对实时性要求不高。因此，Hive 并不能够在大规模数据集上实现低延迟、快速的查询。例如，Hive 在几百 MB 的数据集上执行查询一般有分钟级的时间延迟。因此，Hive 并不适合那些需要低延迟的应用，例如，联机事务处理（OLTP）。

Hive 的查询操作过程严格遵守 Hadoop MapReduce 的作业执行模型。Hive 将用户的 HiveQL 语句通过解释器转换为 MapReduce 作业提交到 Hadoop 集群上，Hadoop 监控作业执行过程，然后返回作业执行结果。Hive 并非为联机事务处理而设计，而且不提供实时的查询和基于行级的数据更新操作。Hive 的最佳使用场合是大数据集的批处理作业，例如，网络日志分析。Hive 的优势在于处理大数据，对于处理小数据没有优势，因为 Hive 的执行延迟比较高。

4.2 大数据仓库 HBase

4.2.1 HBase 因何而生

首先需要搞清楚 HBase 是什么。HBase 是 Hadoop Database（Hadoop 数据库）的简写。官方对 HBase 的定义是：HBase 是一种高可靠性、高性能、面向列、可

伸缩的分布式存储系统，利用 HBase 技术可在廉价的 PC Server 上搭建起大规模的结构化存储集群。

在上面的官方定义中，排除一些“冠冕堂皇”的客套话，我们发现了至关重要的一点：面向列。这是 HBase 相对于传统的数据库来说最为精华的东西。那么，如何理解这个“面向列”呢？

如果有“面向列”，那就有“面向行”了？是的。在传统的数据库中，数据是面向行存储的，数据库表的每一行都是一条独立的数据，存储在磁盘上的一块区域内，通过标记哪一列是什么值来存储。我们举一个简单的例子，例如表 4-1 是一张数据库表。

表 4-1 数据库表示例

编　号	名　称	性　别	男 朋 友	女 朋 友
001	王二小	男		王小丫
002	张大花	女	张大宝	
003	李二牛	男	王麻子	

面向行的数据库，一行是一个完整的数据，比如在表 4-1 中就有三行完整的数据。面向行的数据库有三个问题：

（1）浪费磁盘空间。如果有一行数据需要插入到数据库中，数据库就会为这行数据根据数据类型来开辟存储空间，这个空间一旦开辟出来，尽管有些字段是空的，那么这个存储空间也会被占用，可能会出现部分存储空间白白浪费的情况。拿表 4-1 来说，当王二小的性别是男时，那么他的“男朋友”一列就是空的；而对于张大花这位女生来说，她的“女朋友”一列就是空的。为了不浪费空间，我们来做个判断：如果性别是男，就不留“男朋友”一列；如果性别是女，就不留“女朋友”一列。且不论技术上的复杂度（有类似的数据库技术，但这么做会影响性能），单从用户需求层面来说，用户的各种需求都是不确定的，如果省略掉某些列，就可能出现错误。

（2）查询浪费磁盘 I/O 和网络传输资源。为什么呢？假如我们输入这样的查询：select (" 编号 "," 名称 ") from tables。要知道，尽管我们只查询编号和名称，但是作为服务器端的返回值，是将满足条件的整个数据反馈在客户端，客户端再进行过滤。上面例子中的数据量比较小，如果数据量很大，则会对磁盘 I/O 和网络传输造成一定的影响，并且客户端还要足够强大才能快速处理反馈的数据。

（3）不够灵活。假如面对非结构化的数据，使用面向行的数据库，则不一定能将数据按照行的要求来分割。比如一个文件，面向行的数据库就显得不够灵活。

原因还有很多，总之，聪明的数据库研发工程师发现，对于大数据处理的环境，面向列的存储能够解决上述问题，并且存储灵活，性能也有很大幅度的提升。“面向列”存储是指不同的列（一列或者多列）存储在不同的磁盘文件中，是按照不同的列簇来存储的。那什么是列簇呢？简单地理解，就是具有某个相同特征的列的集合。我们后续还会对这个概念进行详细讲述。

其实从某种层面上讲，HBase 是 NoSQL 数据库的一种。那什么是 NoSQL 数据库呢？ NoSQL 数据库是非关系型数据存储的统称。NoSQL 数据存储不需要固定的表结构，通常也不存在连接操作，在大数据存储方面具有关系型数据库无法比拟的性能优势。随着 Web 2.0 的兴起，NoSQL 数据库成了一个极其热门的新领域，NoSQL 产品的发展非常迅速。那么 NoSQL 数据库都有哪些呢？除 HBase 以外，目前有 BigTable、Cassandra、Hypertable、MongoDB、Redis、CouchDB 等。目前 NoSQL 数据库的种类可能在 100 种以上，那么为什么 NoSQL 数据库发展得如此之快呢？主要原因是关系型数据库有几个天生的限制，使其很难满足用户需求。

（1）扩展困难：由于存在类似 Join 的多表查询机制，使得数据库在扩展方面很艰难。

（2）读 / 写慢：在数据量达到一定规模时，由于关系型数据库的系统逻辑非常复杂，使得其非常容易发生死锁等并发问题，导致其读 / 写速度下滑非常严重。

（3）成本高：企业级数据库的 License 价格惊人，并且随着系统规模的增加而不断上升。

（4）有限的支撑容量：现有的关系型解决方案还无法支撑 Google 这样的海量数据存储。

业界为了解决上述问题，推出了多款 NoSQL 系列数据库。总的来说，在设计上，它们非常关注对数据高并发地读 / 写和对海量数据的存储等；与关系型数据库相比，它们在架构和数据模型方量面做了“减法”，而在扩展和并发等方面做了“加法”。NoSQL 系列数据库的优势主要体现在以下三点。

（1）简单的扩展：典型例子是 Cassandra，由于其架构类似于经典的 P2P，所以能通过轻松地添加新的节点来扩展这个集群。

（2）快速的读 / 写：典型例子是 Redis，由于其逻辑简单，而且纯内存操作，使得其性能非常出色，单节点每秒可以处理超过 10 万次读 / 写操作。

（3）低廉的成本：这是大多数分布式数据库共有的特点，因为它们大部分是开源软件，没有昂贵的 License 成本。

而 HBase 正是其中的一员，由 Apache 基金会开发。HBase 可以横向扩张的表存储系统能够为大规模数据提供速度极快的低等级更新，主要用于需要随机访问、实时读 / 写大数据（Big Data）的情形。这正是信息系统所需要的功能。

4.2.2 HBase 的设计思想和架构

1. HBase 的重要概念

我们在前面反复阐述，HBase 是以列的形式存储数据的。举个简单的例子，我们将 HBase 的数据的逻辑存储模型通过表 4-2 展示。

表 4-2 HBase 的数据的逻辑存储模型

Row Key	column-family1		column-family2		column-family3	
	column1	column2	column1	column2	column3	column1
key1	t1:abc t2:gdxdf		t4:dfads t3:hello t2:world			
key2	t3:abc t1:gdxdf		t4:dfads t3:hello		t2:dfdsfa t3:dfdf	
key3		t2:dfadfasd t1:dfdasddsf				t2:dfxxdfasd t1:taobao.com

1）列簇（column-family）

列簇是相关列的集合。HBase 中的每一列都属于某一个列簇，列的命名与其所属的列簇有关，一般以列簇作为前缀，例如 family1.test1、family1.test2 都属于 family1 这个列簇。访问控制、磁盘和内存的使用统计都是在列簇层面进行的。在实际应用中，列簇上的控制权限能帮助我们管理不同类型的应用：我们允许一些应用添加新的基本数据，一些应用读取基本数据并创建继承的列簇，一些应用则

只能浏览数据（甚至可能因为隐私而不能浏览所有数据）。在表 4-2 中有三个列簇，每个列簇又包含不同的列。

2）行键（Row Key）

类似于关系型数据库里面的主键，用来检索记录。可以通过行键“得到”行，或者通过给定起始和终止行键检索一系列行。行键可以是任意字符串（最大长度是 64KB，实际应用中长度一般为 10～100Bytes）。在 HBase 内部，行键保存为字节数组。在存储时，数据按照行键的字典序（Byte Order）排序存储。在设计 key 时，要充分利用排序存储这个特性，将经常一起读取的行存储在一起。如表 4-2 所示，key1,key2,key3 是三条记录的唯一 Row Key 值。每个行键存储在 HDFS 上的一个单独文件中，空值不会被保存。

3）单元（Cell）

HBase 中通过 row 和 column 确定的一个存储单元称为 Cell。Cell 唯一标识的表达形式为 {row key, column(=<family> + <label>), version}。Cell 中的数据是没有类型的，全部采用字节码形式存储。

4）时间戳（Timestamp）

每个 Cell 都保存着同一份数据的多个版本，版本通过时间戳来索引。时间戳的类型是 64 位整型。时间戳可以由 HBase（在数据写入时自动）赋值，此时时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显式赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。在每个 Cell 中，不同版本的数据按照时间倒序排序，即最新的数据排在最前面。

为了避免数据存在过多版本造成的管理（包括存储和索引）负担，HBase 提供了两种数据版本回收方式：一种是保存数据的最后 *n* 个版本；另一种是保存最近一段时间内的版本（比如最近七天）。用户可以针对每个列簇进行设置。

比如 column-family1 这个列簇下包括两列，分别是 column1 和 column2，t1:abc,t2:gdxdf 是由 row key1 和 column-family1-column1 唯一确定的一个单元。这个单元中有两个数据：abc 和 gdxdf。这两个数据的时间戳不一样，分别是 t1 和 t2，HBase 会返回最新时间的值给请求者。

5）Region（区域）

这是 HBase 中至关重要的概念。细心的读者可能会发现，笔者将中文翻译写到了括号里面，因为在大多数情况下，人们更喜欢用英文来称呼，那么下文我们也会用英文来表达。

前面已经提到，Table 中的所有行都按照行键的字典序排列，Table 在行的方向上分割为多个 Region，如图 4-5 所示。

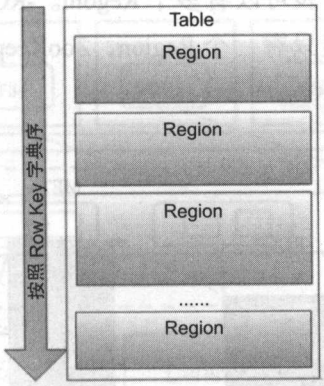


图 4-5　HBase 中的 Table

Region 是 HBase 中分布式存储和负载均衡的最小单元，不同的 Region 分布到不同的 Region Server 上。Region 是按大小分割的，每张表一开始只有一个 Region，随着数据增多，Region 不断增大，当增大到一个阈值的时候，Region 就会等分为两个新的 Region，之后会有越来越多的 Region，如图 4-6 所示。

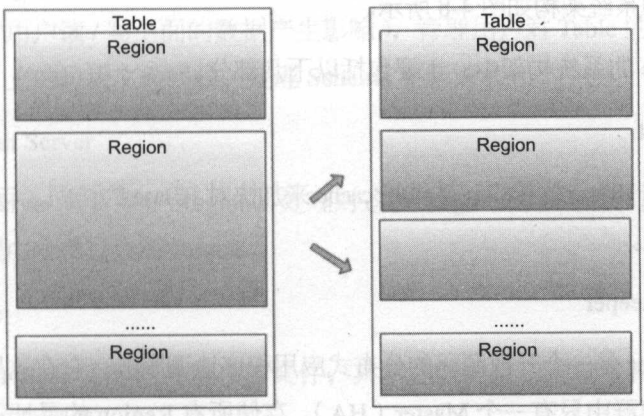


图 4-6　Region 的增加示意图

HBase Region 虽然是分布式存储的最小单元，但并不是存储的最小单元。事实上，HBase Region 由一个或者多个 Store 组成，每个 Store 保存一个 column-family。每个 Store 又由一个 memStore 和 0 至多个 StoreFile 组成。StoreFile 以 HFile 格式保存在 HDFS 上。

6）HBase 中两张特殊的表

-ROOT- 表和 .META. 表是 HBase 中两张特殊的表。.META. 表记录了用户表的 Region 信息，.META. 表可以有多个 Regoin。-ROOT- 表记录 了 .META. 表的 Region 信息，-ROOT- 表只有一个 Region，ZooKeeper 中记录了 -ROOT- 表的 Location。具体如图 4-7 所示。

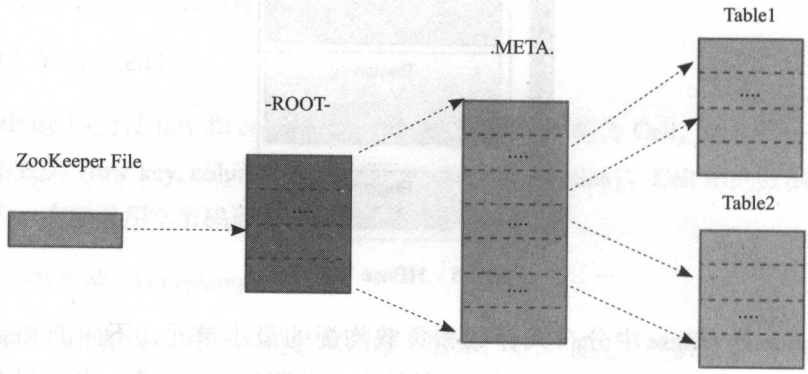


图 4-7　HBase 中两张特殊的表

2. HBase 系统架构

HBase 的系统架构如图 4-8 所示。

在 HBase 的系统构架中，主要包括以下几部分。

1）Client

包含访问 HBase 的接口，并维护 cache 来加快对 HBase 的访问，与 HBase Region Server 打交道。

2）ZooKeeper

ZooKeeper 是一个开放源码的分布式应用程序协调服务，它在这里的作用是保证任何时候集群中只有一个 Master（HA），存储所有 Region 的寻址入口，实时监

控 Region Server 的上线和下线信息并实时通知 Master，存储 HBase 的 Schema 和 Table 元数据。

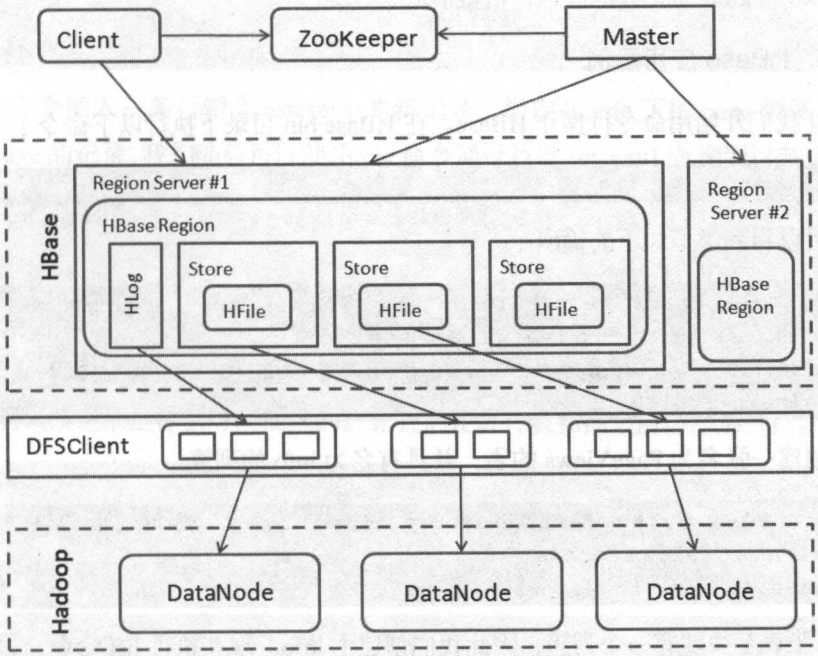


图 4-8　HBase 的系统架构

3）Master

Master 的作用是为 Region Server 分配 Region，负责 Region Server 的负载均衡，发现失效的 Region Server 并重新分配其上的 Region 到其他 Region Server（需要时间，但是会对用户读 / 写上面的数据产生影响），管理用户对 Table 的增、删、改、查操作，GFS 上的垃圾文件回收，处理 Schema 更新请求。

4）Region Server

Region Server 负责维护 Region，处理对这些 Region 的 I/O 请求，同时负责切分在运行过程中变得过大的 Region。

5）HFile

HFile 是 HBase 实际的物理存储文件，其中存放的是二进制数据，类似 Word 文档或者 TXT 文件等。

可以看到，Client 访问 HBase 上的数据并不需要 Master 参与（寻址访问 ZooKeeper 和 Region Server，数据读 / 写访问 Region Server），Master 仅仅维护 Table 和 Region 的元数据信息，负载很低。

3. HBase 应用示例

让我们开始用命令行操作 HBase。在 HBase bin 目录下执行以下命令：

```
./hbase shell
```

可以看到类似如下的输出：

```
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.5-hadoop2, rUnknown, Mon Aug  4 23:58:06 PDT 2014
hbase(main):001:0>
```

创建一张名为 PageViews 的表，并具有名为 info 的列簇。

```
hbase(main):002:0> create 'PageViews', 'info'
0 row(s) in 5.3160 seconds
=> Hbase::Table - PageViews
```

每张表至少要有一个列簇，因此我们创建了 info。现在执行 list 命令。

```
hbase(main):002:0> list
TABLE
PageViews
1 row(s) in 0.0350 seconds
=> ["PageViews"]
```

可以看到，list 命令返回一张名为 PageViews 的表。我们可以通过 describe 命令得到表的更多信息。

```
hbase(main):003:0> describe 'PageViews'
DESCRIPTION
'PageViews', {NAME => 'info', DATA_BLOCK_ENCODING => 'NONE',
BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1',
COMPRESSION => 'NONE true', MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_
DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'false',
BLOCKCACHE => 'true'}
1 row(s) in 0.0480 seconds
```

describe命令返回表的详细信息，包括列簇的列表。这里我们创建的列簇仅有一个，即info，现在向info中添加新的行。

```
hbase(main):004:0> put 'PageViews', 'rowkey1', 'info:page', '/mypage'
0 row(s) in 0.0850 seconds
```

put命令插入一条行键为rowkey1的新记录，指定在info下的page列插入值为/mypage的记录，我们随后可以使用get命令通过行键rowkey1查询到这条记录。

```
hbase(main):005:0> get 'PageViews', 'rowkey1'
COLUM              CELL
info:page            timestamp=1410374788088, value=/mypage
1 row(s) in 0.0250 seconds
```

我们可以看到列info:page，或者更多具体的列，其值为/mypage，并带有时间戳，表明该条记录是什么时候插入的。让我们在进行表扫描之前再添加一行。

```
hbase(main):006:0> put 'PageViews', 'rowkey2', 'info:page', '/myotherpage'
0 row(s) in 0.0050 seconds
```

现在我们有两行记录了。让我们查询出PageViews表的所有记录。

```
hbase(main):007:0> scan 'PageViews'
ROW    COLUMN+CELL
rowkey1      column=info:page, timestamp=1410374788088, value=/mypage
rowkey2   column=info:page, timestamp=1410374823590, value=/myotherpage
2 row(s) in 0.0350 seconds
```

如前面所提到的，我们不能查询表本身，但是我们可以对表进行scan（扫描）操作。如果你执行scan table命令，则会返回表中的所有行，这很有可能不是你想要的。你可以给出行的范围来限制返回的结果。让我们插入一个以s开头的行键的新记录：

```
hbase(main):012:0> put 'PageViews', 'srowkey2', 'info:page', '/myotherpage'
```

现在，如果增加点限制，想查询行键在r和s之间的记录，则可以使用如下结构：

```
hbase(main):014:0> scan 'PageViews', { STARTROW => 'r', ENDROW => 's' }
ROW         COLUMN+CELL
rowkey1   column=info:page, timestamp=1410374788088, value=/mypage
rowkey2   column=info:page, timestamp=1410374823590, value=/myotherpage
2 row(s) in 0.0080 seconds
```

这个 scan 返回了仅以 s 开头的记录，这个类比是基于全行键上的，因为 rowkey1 比 r 大，所以它被返回了。另外，scan 的结果包含了所指范围的 STARTROW，但不包含 ENDROW。注意，ENDROW 不是必须指定的。如果我们执行相同的查询但只给出了 STARTROW，那么我们会得到行键比 r 大的所有记录。

```
hbase(main):013:0> scan 'PageViews', { STARTROW => 'r' }
ROW     COLUMN+CELL
rowkey1  column=info:page, timestamp=1410374788088, value=/mypage
rowkey2  column=info:page, timestamp=1410374823590, value=/myotherpage
srowkey2 column=info:page, timestamp=1410375975965, value=/myotherpage
3 row(s) in 0.0120 seconds
```

4.2.3 HBase 优化技巧

1. 修改 Linux 系统参数

Linux 系统最大可打开文件数一般默认是 1024，如果不进行修改，则并发量增大的时候可能会出现“Too Many Open Files”错误，导致整个 HBase 不可运行。可以采用 ulimit -n 命令进行修改，或者修改 /etc/security/limits.conf 和 /proc/sys/fs/file-max 的参数，具体如何修改请读者查询关键字“linux limits.conf”。

2. JVM 配置

修改 hbase-env.sh 文件中的配置参数。可根据计算机硬件和当前操作系统的 JVM（32/64 位）配置适当的参数。

```
HBASE_HEAPSIZE 4000          //HBase使用的JVM堆的大小
HBASE_OPTS "-server -XX:+UseConcMarkSweepGC"//JVM GC选项
HBASE_MANAGES_ZKfalse        //是否使用ZooKeeper进行分布式管理
```

3. HBase 持久化

重启操作系统后 HBase 中的数据全无。你可以在不进行任何修改的情况下创建一张表，写入一条数据，然后重启计算机，再次进入 HBase 的 Shell 中，使用 list 命令查看当前存在的表，你会发现一张表都没有了。不要着急，我们可以在 hbase/conf/hbase-default.xml 文件中设置 hbase.rootdir 的值，来为文件的保存位置指定一个文件夹，例如 <value>file:///you/hbase-data/path</value>，这样 HBase 中的表和数据就会直接写到磁盘上。同样也可以指定分布式文件系统 HDFS 的路径，例如 hdfs://

NAMENODE_SERVER:PORT/HBASE_ROOTDIR，这样 HBase 中的表和数据就会写到分布式文件系统上。

4. 配置 HBase 运行参数

我们还需要对 hbase/conf/hbase-default.xml 文件进行配置，以下是比较重要的配置参数。

- hbase.client.write.buffer：这个参数可以设置写入数据缓冲区的大小。当客户端和服务器端传输数据时，服务器为了提高系统运行性能，会开辟一个写的缓冲区来进行处理。这个参数如果设置得太大，将会对系统的内存有一定的要求，直接影响系统的性能。
- hbase.master.meta.thread.rescanfrequency：多长时间 HMaster 对系统表 root 和 meta 扫描一次。这个参数可以设置得大一些，以降低系统的能耗。
- hbase.regionserver.handler.count：由于 HBase/Hadoop 的 Server 是采用 Multiplexed, non-blocking I/O 方式而设计的，所以它可以通过一个 Thread 来完成处理，但是由于处理客户端所呼叫的方法是 Blocking I/O，所以它的设计会将客户端传递过来的物件先放置在 Queue，并在启动 Server 时先产生一堆 Handler（Thread），该 Handler 会通过 Polling 的方式来取得该物件并执行对应的方法。默认值为 25，根据实际场景可以设置得大一些。
- hbase.regionserver.thread.splitcompactcheckfrequency：RegionServer 服务器运行一次 split/compaction 的时间间隔，当然在 split 之前会先进行一次 compact 操作。这个 compact 操作可能是 minor compact，也可能是 major compact.compact。然后会从 Store 下所有 StoreFile 文件中最大的那个取 midkey。这个 midkey 可能并不处于全部数据的 mid 中。一个 row-key 下的数据可能会跨不同的 HRegion。
- hbase.hregion.max.filesize：HRegion 中的 HStoreFile 最大值，任何表中的列簇一旦超过这个值将会被切分。HStoreFile 的默认大小是 256MB。
- hfile.block.cache.size：指定 HFile/StoreFile 缓存在 JVM 堆中分配的百分比，默认值是 0.2，意思就是 20%。如果设置为 0，则表示对该选项进行屏蔽。
- hbase.zookeeper.property.maxClientCnxns：这项配置的选项就是从 ZooKeeper

中来的，表示 ZooKeeper 客户端同时访问的并发连接数。ZooKeeper 对于 HBase 来说就是一个入口，这个参数的值可以适当放大些。

- hbase.regionserver.global.memstore.upperLimit：在 Region Server 中所有 MemStore 占用堆的大小，默认值是 0.4，表示 40%。如果设置为 0，则表示对该选项进行屏蔽。
- hbase.hregion.memstore.flush.size：MemStore 中缓存的内容超过配置的范围后将会写到磁盘上。例如，删除操作是先写入 MemStore 里做个标记，指示哪个 value、column 或 family 是要删除的，HBase 会定期对存储文件做一个 major compaction，到那时 HBase 会把 MemStore 刷入一个新的 HFile 存储文件中。如果在一定时间范围内没有做 major compaction，则 MemStore 中超出的范围就会写到磁盘上。

5. HBase 中 log4j 的日志

HBase 中日志输出等级默认状态下是把 debug、info 级别的日志打开的，可以根据自己的需要调整 log 级别。HBase 的 log4j 日志配置文件存放在 hbase\conf\log4j.properties 目录下。

4.2.4 HBase 和 Hive 的区别

HBase 和 Hive 具有如下区别：

（1）HBase 是低延迟、非结构化和面向编程的，而 Hive 是高延迟、结构化和面向分析的。

（2）Hive 本身不存储和计算数据，它完全依赖于 HDFS 和 MapReduce。Hive 中的表是逻辑表。

（3）HBase 通过组织起节点内所有机器的内存，提供一张超大的内存 Hash 表，它需要在磁盘和内存中组织自己的数据结构。HBase 中的表是物理表。

（4）如果是全表扫描，就用 Hive+Hadoop；如果是索引访问，就用 HBase+Hadoop。

（5）Hive 主要用于静态的结构及需要经常分析的工作。

（6）Hive 可以与 HBase 组合使用，Hive 为 HBase 提供了高层语言支持，使得在 HBase 上进行数据统计处理变得非常简单。

4.3 Pig 编程语言

4.3.1 Pig 的缘由

Pig 是基于 Hadoop 的大规模数据分析平台，也可以理解为一门高级语言，它提供的 SQL-like 语言称为 Pig Latin，该语言的编译器会把类 SQL 的数据分析请求转换为一系列经过优化处理的 MapReduce 运算，适合于使用 Hadoop 和 MapReduce 平台来查询大型半结构化数据集。Pig 最早是雅虎公司的一个基于 Hadoop 的并行处理架构，后来雅虎将其迁移到 Apache 来维护，作为 Hadoop 阵营中的一个子项目。那么，当初雅虎公司主要使用 Pig 来干什么呢？

（1）吸收和分析用户的行为日志数据（点击流分析、搜索内容分析等），改进匹配和排名算法，以提高检索和广告业务的质量。

（2）构建和更新 search index。需要处理的数据形式为流数据，这包括去冗余、链接分析、内容分类、基于点击次数的受欢迎程度计算（PageRank）、建立倒排表。

（3）处理半结构化数据订阅（Data Seeds）服务，包括 deduplcaitin（去冗余）、geographic location resolution 及 named entity recognition。

目前看来，Pig 的主要开发者和贡献者不止是雅虎，还有 Hortonworks、Twitter、Cloudera 等。那么，为什么有这么多的公司都参与到 Pig 的开发中来呢？我们知道，运行 Hadoop MapReduce 应用，在未借助其他工具的情况下，如果手写 MapReduce 代码，则是一件非常烦琐的事。因为 MapReduce 的任务职责非常明确，读 / 写数据得一个 Job，处理得一个 Job，过滤得一个 Job，统计得一个 Job，排序得一个 Job，编写 DAG（带先后顺序依赖的）作业很不方便。这还可以接受，但是每次只要改动很小的一个地方就得重新编译整个 Job，然后打包成 JAR 包提交到 Hadoop 集群上运行，不仅非常烦琐，而且调试还很困难。所以，现在的大型互联网公司或者电商很少有纯写 MapReduce 来处理各种任务的，基本上都会使用一些工具或开源框架来操作。Pig 就是这样的工具，使用 Pig 来操作 Hadoop 处理海量数据是非常简单的。

有了 Pig 之后，就不再需要编写一个单独的 MapReduce 应用程序，可以用 Pig

Latin 语言编写一个脚本，在集群中自动并行处理与分发该脚本。让我们从一个简单的 Pig 示例开始介绍，并剖析该示例。Hadoop 的一个有趣用法是，在大型数据集中搜索满足某个给定搜索条件的记录（在 Linux 中被称为 grep）。下面这段代码显示了在 Pig 中实现该过程的简单性。

```
messages = LOAD 'messages';
warns = FILTER messages BY $0 MATCHES '.*WARN+.*';
STORE warns INTO 'warnings';
```

在上述三行代码中，只有一行是真正的搜索。第一行只是将测试数据集（消息日志）读取到代表元组集合的包中。首先用一个正则表达式来筛选该数据（元组中的唯一条目，表示为 $0 或 field 1），然后查找字符序列 WARN，最后在主机文件系统中将这个包存储在一个名为 warnings 的新文件中。这个包现在代表来自消息的包含 WARN 的所有元组。

这个简单的脚本实现了一个简单的流，但是，如果直接在传统的 MapReduce 模型中实现它，则需要增加大量的代码。这使得学习 Hadoop 并开始使用数据比原始开发容易得多。

随着数据海啸的来临，传统 DB（Oracle、DB2）已经不能满足海量数据处理的需求，MapReduce 逐渐成为数据处理的事实标准，被应用到各行各业中。所以，我们不再期望所有的客户都能快速开发应用相关代码，只能把客户的工作变得简单，就像使用 SQL 语言，经过简单培训就可以在“云”上操作。Pig 就是为了屏蔽 MapReduce 开发的烦琐细节，为用户提供 Pig Latin 这样的近 SQL 语言处理能力，让用户可以更方便地处理海量数据。Pig 将 SQL 语句翻译成 MapReduce 作业的集合，并通过数据流的方式将其组合起来。

Pig 的优点在于：编码简单；对常见的操作充分优化；可扩展；自定义 UDF。所以，Pig 的应用场景一般有：数据查询只面向相关技术人员；即时性的数据处理需求，这样可以通过 Pig 快速编写一个脚本开始运行处理，而不需要创建表等相关的事前准备工作。

4.3.2 Pig 的基本架构

Pig 系统由 5 个主要部分构成，如图 4-9 所示。

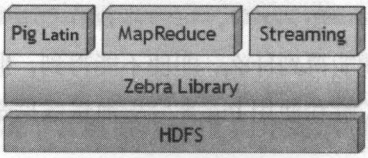


图 4-9 Pig 的基本架构

（1）Pig Latin 是面向数据流的编程方式，而 SQL 是一种描述型编程语言。我们以前学习 SQL 的时候经常听到这样一句话：用 SQL，你只需要告诉它你需要什么，具体怎么做交给 SQL 就行了。而 Pig Latin 是需要你一步一步地根据数据流的处理方式来编程的，也就是说你要设计数据流的每一个步骤，有点类似 SQL 的查询规划器。

（2）Zebra Library 是 Pig 与 HDFS/Hadoop 的中间层，也是 MapReduce 作业编写的客户端，它用结构化的语言实现了对 Hadoop 物理存储元数据的管理。Zebra Library 中有两个核心的类 TableStore（写）/TableLoad（读）对 Hadoop 上的数据进行操作。

（3）Pig 中的 Streaming 主要分为 4 个组件：Pig Latin；逻辑层（Logical Layer）；物理层（Physical Layer）；Streaming 具体实现（Implementation）。Streaming 会创建一个 Map/Reduce 作业，并把它发送给合适的集群，同时监视这个作业在集群环境中的执行过程。

（4）MapReduce：在每台机器上进行分布式计算的框架（算法）。

（5）HDFS：最终存储数据的部分。

Pig 的执行引擎如图 4-10 所示。

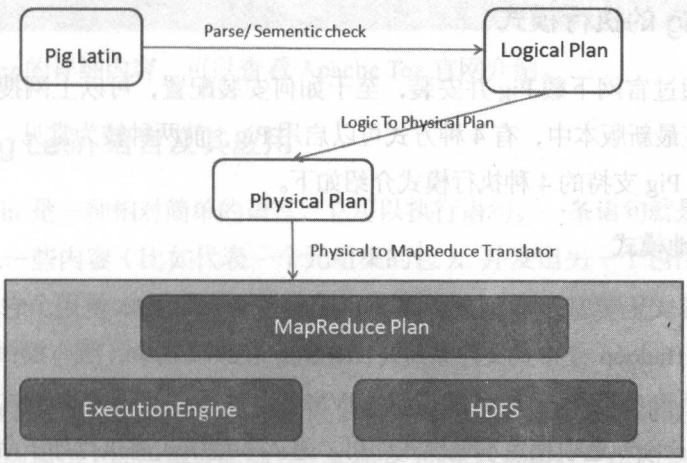


图 4-10 Pig 的执行引擎

在 Pig 中，每一步操作都是一个数据流。这非常容易理解，你想要什么，你就能得到什么，即使不能得到，也可以通过轻松扩展 UDF 来实现。在大数据时代，了解和使用 Pig 来分析海量数据是非常容易的。

4.3.3 Pig 与 Hive 的对比

Pig 为复杂的海量数据并行计算提供了一个简易的操作和编程接口，这一点和 Facebook 开源的 Hive 一样简洁、清晰、易上手。

- Language：在 Hive 中可以执行“插入”、“删除”等操作，但在 Pig 中没有可以“插入”数据的方法。
- Schemas：Hive 中至少还有一个“表”的概念，但在 Pig 中基本没有表的概念。所谓的表建立在 Pig Latin 脚本中，对于 Pig 更不要提 MetaData 了。
- Partition：Pig 中没有表的概念，所以说到分区对于 Pig 来说基本没有意义。如果跟 Hive 说“分区”（Partition），它还是能明白的。
- Server：Hive 可以依托 Thrift 启动一台服务器，提供远程调用，但在 Pig 中没有发现类似的功能。
- Shell：在 Pig 中可以执行 ls、cat 等经典命令，但是在使用 Hive 的时候基本没有这样的需求。
- Web Interface：Hive 有，Pig 无。
- JDBC/ODBC：Pig 无，Hive 有。

4.3.4 Pig 的执行模式

可以通过官网下载 Pig 并安装，至于如何安装配置，可以上网搜索，这里不再阐述。在最新版本中，有 4 种方式可以启用 Pig，前两种较为常见，后两种尚在探索阶段。Pig 支持的 4 种执行模式介绍如下。

1. 本地模式

本地模式主要基于本地文件系统，比较适合调试脚本使用。它不需要依赖 Hadoop 或 Hadoop 分布式文件系统（HDFS）。在该模式中，所有操作都在本地文件系统上下文中的单一 Java 虚拟机（JVM）上执行。对于本地模式，只需启动 Pig 并用 exectype 选项指定本地模式即可。进入本地模式需执行如下命令：

```
pig -x local
```

2. MapReduce 模式

MapReduce 模式是基于 Hadoop 的。一般情况下，数据存储在 HDFS 上，使用该模式能够充分利用 Hadoop 集群的计算能力，它基于运行于 YARN 之上的 MapReduce 进行处理。对于 MapReduce 模式，必须首先确保 Hadoop 正在运行。要做到这一点，最简单的方法是在 Hadoop 文件系统树的根上执行文件列表操作。进入 MapReduce 运行模式需执行如下命令：

```
pig -x mapreduce
```

3. Tez 本地模式

Tez 本地模式类似于前面的本地模式，它使用 Tez 运行时引擎。进入 Tez 本地模式需执行如下命令：

```
pig -x tez_local
```

不过该模式还处于试验阶段，不过多叙述。

4. Tez 模式

基于 Tez 模式执行，需要在安装 Hadoop 集群时修改 Hadoop 配置文件 mapred-site.xml，将属性 mapreduce.framework.name 的值设置为 yarn-tez。进入 Tez 模式需执行如下命令：

```
pig -x tez
```

有关 Tez 的详细内容，可以查看 Apache Tez 官网介绍。

4.3.5 Pig Latin 语言及其应用

Pig Latin 是一种相对简单的语言，它可以执行语句。一条语句就是一个操作，它需要输入一些内容（比如代表一个元组集的包），并发出另一个包作为其输出。一个包就是一个关系，与表类似，可以在关系数据库中找到它（其中，元组代表行，并且每个元组都由字段组成）。

用 Pig Latin 编写的脚本往往遵循以下特定格式：从文件系统读取数据，对数据执行一系列操作（以一种或多种方式转换它），然后将由此产生的关系写回文件系统。

1. 数据类型

Pig 拥有大量的数据类型，不仅支持包、元组和映射等高级概念，还支持简单的数据类型，大致可以分为两类，分别为简单类型和复杂类型。

简单类型包括：int、long、float、double、chararray、bytearray、boolean、datetime、biginteger、bigdecimal。其中有几个看起来很是稀奇古怪，比如 chararray 和 bytearray，其实 chararray 相当于字符串 String，bytearray 相当于字节数组。

复杂类型包括：tuple、bag、map。tuple 是一个有序的字段的集合，可以理解为元组，例如 (440302,'google',30)；bag 是 tuple 的集合，例如 {(440302,'google',30),(40302,'BING',30)}；map 是键值对的集合，例如 [name#xiaoming, age#30,healthy index#99.58]。

2. 基本操作符

- 算术操作符（Arithmetic Operators）包括：+、-、*、/、%、?:、CASE WHEN THEN ELSE END。
- 布尔操作符（Boolean Operators）包括：AND、OR、IN、NOT。
- 类型转换操作符（Cast Operators）：使用圆括号包含类型名，作用于一个字段，例如 (int)age、(map[])、(chararray)COUNT($2)、(tuple(chararray, int, map[])) name_age_scores 等。
- 比较操作符（Comparison Operators）包括：==、!=、=、matches。其中，matches 比较操作符使用 Java 的 Pattern 进行匹配来比较，例如 user_name matches '[a-n]{3,12}'。
- 类型构造操作符（Type Construction Operators）：可以创建复杂类型的数据，tuple 使用 ()，map 使用 []，bag 使用 {}。例如 FOREACH users GENERATE (name, age, address)。
- 解引用操作符（Dereference Operators）：解引用主要针对集合类型 tuple、bag、map，从集合中拿到对应字段的值。比如对于 tuple，定义类型 t=tuple(t1:int,t2:int,t3:int)，若要获取字段 t1 和 t3 的值，可以通过 t.t1 和 t.t3 得到，也可以通过 t.$0 和 t.$2 得到。

3. Pig 应用示例

1）ASSERT

```
live_user_ids=LOAD '/test/live_user_ids' USING PigStorage() AS (udid: chararray);
ASSERT live_user_ids BY udid != null, 'udid MUST NOT be NULL!';
```

上述语句断言表 live_user_ids 中的 udid 字段一定存在值。

2）GROUP

根据某个或某些字段进行分组。只根据一个字段进行分组比较简单。如果想要根据两个字段分组，则可以将两个字段构造成一个 tuple，然后进行分组。Pig 脚本如下：

```
events = LOAD '/data/etl/hive_input/20150613/basis_event_2015061306-r-00002'
USING PigStorage('\t');
projected_events = FOREACH events GENERATE $1 AS (event_code: long),
$2 AS (udid: chararray), $10 AS (network: chararray), $34 AS (area_code:
int); -- $1是表events的第二个字段
uniq_events = DISTINCT projected_events;
uniq_events = FILTER uniq_events BY (event_code IS NOT NULL) AND (udid IS
NOT NULL) AND (network IS NOT NULL) AND (area_code IS NOT NULL);
groupped = GROUP uniq_events BY (udid, event_code) PARALLEL 2; -- 指定使
用两个Reducer
selected10 = LIMIT groupped 10;
DUMP selected10;
```

从 HDFS 加载的文件中执行投影操作，生成包含 event_code、udid、network、area_code 这 4 个字段的一张表 projected_events，接着执行去重、条件过滤操作，计算分组的时候基于 event_code 和 udid 两个字段。

3）FOREACH

FOREACH 操作可以针对一个数据集进行迭代处理，生成一个新的数据集。它有两种使用方法，一种是执行投影操作，选择部分字段的数据，示例脚本如下：

```
provinces = LOAD '/test/provinces' USING PigStorage(',') AS (country_id:
int, province_id: int, name: chararray);
compositekeyed_provinces = FOREACH provinces GENERATE (CONCAT(CONCAT((chararray)
country_id, '_'), (chararray)province_id) AS pid, name);
```

这里将原数据集的两个主键字段的值进行拼接合并，作为新表的一个字段。

另一种是支持在 FOREACH 操作中使用代码段，可以增加更复杂的处理逻辑。

4）FILTER

根据条件进行过滤，相当于 SQL 中的 WHERE 子句。示例脚本如下：

```
live_info = LOAD '/test/live_info' USING PigStorage() AS (id: long, name:
chararray);
newly_added_lives = FILTER live_info BY (id % 140000 >= 0) AND (name
matches '[a-zA-Z0-9]{8, 32}' OR name == 'test');
```

上述代码很容易理解，不再累述。

5）JOIN

JOIN 操作支持配置并行度，指定 Reducer 的数量。表连接操作支持内连接和外连接。内连接脚本示例如下：

```
live = LOAD '/test/shiyj/pig/pig_live' USING PigStorage('\t') AS (id:
long, name: chararray);
program = LOAD '/test/shiyj/pig/pig_live_program' USING PigStorage
('\t') AS (id: long,name: chararray,live_id: long,live_start:
chararray,live_end: chararray);
program_info = JOIN live BY id, program BY live_id;
DUMP program_info;
```

上述代码根据表 live 的 id 字段、表 program 的 live_id 字段进行内连接。

关于外连接，举例如下。

左外连接的例子如下：

```
A = LOAD 'a.txt' AS (n:chararray, a:int);
B = LOAD 'b.txt' AS (n:chararray, m:chararray);
C = JOIN A by $0 LEFT OUTER, B BY $0; -- 表A和B的字段n进行左外连接
```

全外连接的例子如下：

```
A = LOAD 'a.txt' AS (n:chararray, a:int);
B = LOAD 'b.txt' AS (n:chararray, m:chararray);
C = JOIN A BY $0 FULL, B BY $0; -- 使用FULL关键字
```

支持复制的左外连接（Replicated Join），示例如下：

```
A = LOAD 'large';
B = LOAD 'tiny';
```

```
C= JOIN A BY $0 LEFT, B BY $0 USING 'replicated';
```

只有左外连接支持这种方式，实际上 Replicated Join 会在 MapReduce 的 Map 阶段把左表进行复制，也就是说左表应该是小表，能够在内存中放得下，然后与右表进行连接操作。还有一种使用 Skewed Join，示例如下：

```
A = LOAD 'studenttab' as (name, age, gpa);
B = LOAD 'votertab' as (name, age, registration, contribution);
C = JOIN A BY name FULL, B BY name USING 'skewed';
```

只有在进行外连接的两张表的数据明显不对称（称为数据倾斜，即一张表很大，另一张表相对较小，但在内存中放不下）时，可以使用 Skewed Join 操作。目前，Pig 支持基于两张表的 Skewed Join 操作。

6）DISTINCT

去重操作使用 DISTINCT 操作符，比较简单，示例如下：

```
live_user_ids = LOAD '/test/live_user_ids' USING PigStorage() AS (udid:
chararray);
uniq_user_ids = DISTINCT live_user_ids PARALLEL 8;
```

DISTINCT 操作支持配置并行度，指定 Reducer 的数量。

7）UNION

计算并集操作使用 UNION 操作符，示例如下：

```
a = LOAD 'data' AS (a1:int,a2:int,a3:int);
b = LOAD 'data' AS (b1:int,b2:int);
u = UNION a, b;
```

计算并集，不要求两张表的字段数一定相同。

8）LIMIT

LIMIT 选择计算结果的一部分，示例如下：

```
top100_user_ids = LIMIT live_user_ids 100;
```

9）STORE

STORE 操作用来保存计算结果，示例如下：

```
STORE play_users INTO '/test/shiyj/tmp.play_users' USING PigStorage ('\t');
```

如果没有指定 USING 子句，则默认使用 PigStorage() 函数。另外，Pig 还支持如下的 Store/Load 函数：

```
BinStorage()
JsonLoader(['schema'])
JsonStorage()
PigDump()
PigStorage([field_delimiter], ['options'])
TextLoader()
HBaseStorage('columns', ['options'])
AvroStorage(['schema|record name'], ['options'])
TrevniStorage(['schema|record name'], ['options'])
AccumuloStorage(['columns'[, 'options']])
OrcStorage(['options'])
```

以上只是举例，Pig 的功能非常强大，没有一本书可以完全列举 Pig 背后处理大数据的强大功能。即使对于非开发人员而言，Pig 也可以使得执行 Hadoop 集群上的大数据处理变得很容易。Pig 最初由雅虎于 2006 年开发，此后不久便被迁移到 Apache 基金，使得它在全球范围得到广泛应用。进行这种迁移是因为雅虎研究人员意识到 Pig 能为非开发人员提供强大的功能。Hadoop 作为一个基础架构已经开始普及，Hadoop 生态系统将会改变大数据的外观及其日益增长的使用情况。

4.4 协管员 ZooKeeper

4.4.1 ZooKeeper 是什么

其实在介绍 HBase 的章节中我们就提到过 ZooKeeper，比较奇怪的名字，看起来不像是某些单词的缩写。其实笔者刚接触的时候也非常好奇，查了字典，ZooKeeper 原来是“动物园管理员”的意思，再结合 ZooKeeper 的功能，便恍然大悟。为什么呢？在 Hadoop 生态圈中有很多“动物”，比如“带头大哥”Hadoop（大象）、数据仓库工具 Hive（蜜蜂的蜂巢）及 Pig（小猪）等，瞬间把 Hadoop 生态圈变成了一个活生生的动物园！

那么这个“动物园管理员”是什么？它的作用又是什么呢？

简单地理解，我们可以认为 ZooKeeper 是一个协管员，是为分布式应用提供

一致性服务的软件。那么它是如何协管的呢？它都解决了 Hadoop 集群中的什么问题呢？ ZooKeeper 主要用来解决分布式应用中经常遇到的一些数据管理问题，如统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。并且 ZooKeeper 包含一个简单的原语集，提供 Java 和 C 的接口。ZooKeeper 的目标就是封装好复杂且易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

4.4.2 ZooKeeper 的作用

我们已经笼统地了解到，ZooKeeper 是用来协助管理 Hadoop Cluster 的。接下来，我们将详细分析 ZooKeeper 的具体作用。

1. ZooKeeper 加强集群稳定性

ZooKeeper 通过一种和文件系统很像的层级命名空间来让分布式进程互相协同工作。这些命名空间由一系列数据寄存器组成，我们称这些数据寄存器为 znodes。这些 znodes 有点像是文件系统中的文件和文件夹。和文件系统不同的是，文件系统的文件是存储在存储区上的，而 ZooKeeper 的数据是存储在内存上的。这就意味着 ZooKeeper 有着高吞吐和低延迟。

ZooKeeper 实现了高性能、高可靠性和有序的访问。高性能保证 ZooKeeper 能应用在大型的分布式系统上；高可靠性保证它不会由于单一节点的故障而造成任何问题；有序的访问能保证客户端可以实现较为复杂的同步操作。

2. ZooKeeper 加强集群持续性

组成 ZooKeeper 的各台服务器必须能够相互通信。它们在内存中保存了服务器状态，也保存了操作的日志，并且持久化快照。只要大多数服务器是可用的，那么 ZooKeeper 就是可用的。

客户端连接到一台 ZooKeeper 服务器，维持 TCP 连接，发送请求，获取回复，获取事件，并且发送连接信号。如果这个 TCP 连接断开，那么客户端可以连接另外一台服务器。

3. ZooKeeper 保证集群有序性

ZooKeeper 使用数字来对每一个更新进行标记，这样就能保证 ZooKeeper 交互

的有序。后续操作可以根据这个顺序实现诸如同步操作这样更高级、更抽象的服务。

4. ZooKeeper 保证集群高效

ZooKeeper 的高效更表现在以读为主的系统上。ZooKeeper 可以在千台服务器组成的读 / 写比例大约为 10 : 1 的分布式系统上表现优异。

5. 数据结构和分等级的命名空间

ZooKeeper 命名空间的结构和文件系统很像。一个名字和文件一样，都使用“/”的路径表现形式，ZooKeeper 的每个节点都被路径唯一标识。

6. ZooKeeper 在 Hadoop 及 HBase 中的具体作用

Hadoop 有 NameNode，HBase 有 HMaster，为什么还需要 ZooKeeper？下面通过例子来介绍。

在一个 ZooKeeper 的集群中，在有 3 个 ZooKeeper 节点、一个 Leader、两个 Follower 的情况下，停掉 Leader，然后两个 Follower 选举出一个 Leader，获取的数据不变。ZooKeeper 能够帮助做到：Hadoop，使用 ZooKeeper 的事件处理机制确保整个集群只有一个 NameNode，存储配置信息等；HBase，使用 ZooKeeper 的事件处理机制确保整个集群只有一个 HMaster，察觉 HRegionServer 联机和宕机，存储访问控制列表等。

4.4.3 ZooKeeper 的架构

ZooKeeper 服务自身组成一个集群（$2n+1$ 个服务允许 n 个失效）。在 ZooKeeper 体系中有 4 个角色：Leader、Follower、Observer、Client。ZooKeeper 的架构如图 4-11 所示。

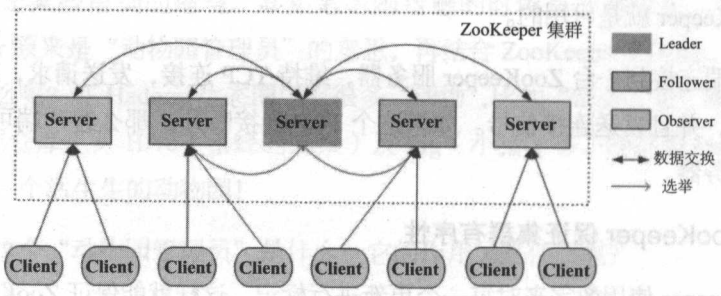


图 4-11　ZooKeeper 的架构

要分析图 4-11，首先需要看看这 4 个角色的作用。

1. Leader

ZooKeeper 需要在所有服务（可以理解为服务器）中选举出一个 Leader，然后让这个 Leader 负责管理集群，进行投票的发起和决议，更新系统的状态等。

2. Follower

此时，集群中的其他服务器则成为此 Leader 的 Follower。并且，当 Leader 出现故障的时候，需要 ZooKeeper 能够快速地在 Follower 中选举出下一个 Leader。那么 Follower 就用于接收 Client 的请求，并向 Client 返回结果，在选举 Leader 的过程中参与投票。

3. Observer

Observer 的行为在大多数情况下与 Follower 完全一致，可以接收客户端的请求，并将写请求转发给 Leader 节点。但在选举 Leader 的过程中，Observer 不参与选举和投票，而仅仅接受选举和投票的结果。那么为什么要设置 Observer 角色呢？其用意在于扩展系统，提高读取速度。

4. Client

Client 就比较简单了，用于发起请求。

在 ZooKeeper 的体系架构中，我们要注意如下几点：客户端可以连接到每个 Server，每个 Server 的数据完全相同；每个 Follower 都和 Leader 有连接，接受 Leader 的数据更新操作；Server 记录事务日志和快照到持久存储；大多数 Server 可用，整体服务就可用。

刚才我们提到，当 Leader 出现故障的时候，需要 ZooKeeper 能够快速地在 Follower 中选举出下一个 Leader。那么如何选举这个 Leader 呢？有两种算法：Basic Paxos 和 Fast Paxos。其大致的核心思路如下：首先创建一个 EPHEMERAL 目录节点，例如“/election”；然后由每台 ZooKeeper 服务器在此目录下创建一个 SEQUENCE|EPHEMERAL 类型的节点，例如“/election/n_”；最后在 SEQUENCE 标志下，ZooKeeper 将自动地为每台 ZooKeeper 服务器分配一个比前一个分配的序号大的序号。此时创建节点的 ZooKeeper 服务器中拥有最小序号编号的服务器将成为 Leader。

在实际操作中还需要保证：当 Leader 服务器发生故障的时候，系统能够快速地选举出下一台 ZooKeeper 服务器作为 Leader。一个简单的解决方案是：让所有的 Follower 监视 Leader 所对应的节点。当 Leader 发生故障时，Leader 所对应的临时节点将会被自动删除，此操作将会触发所有监视 Leader 的服务器的 watch。这样这些服务器将会收到 Leader 故障的消息，并进行下一次的 Leader 选举操作。但是，这种操作将会导致“从众效应”的发生，尤其是当集群中服务器众多并且带宽延迟比较大的时候，此种情况更为明显。

在 ZooKeeper 中，为了避免“从众效应”的发生，它是这样来实现的：每一个 Follower 对 Follower 集群中对应的比自己节点序号小一号的节点（也就是所有序号比自己小的节点中的序号最大的节点）设置一个 watch。只有当 Follower 所设置的 watch 被触发的时候，它才进行 Leader 选举操作，一般情况下它将成为集群中的下一个 Leader。很明显，此 Leader 选举操作的速度是很快的，因为每一次 Leader 选举几乎只涉及单个 Follower 的操作。

4.4.4 ZooKeeper 的数据模型

ZooKeeper 的结构类似标准的文件系统，但这个文件系统中没有文件和目录，而是统一使用节点（node）的概念，称为 znode。这里的 node 不要理解成 Cluster 中的 node（机器实体），实际上，每个子目录项都被称作 znode，这个 znode 被它所在的路径唯一标识。znode 可以有子节点目录，并且每个 znode 可以存储数据（但是 EPHEMERAL 类型的目录节点不能有子节点目录）。znode 作为保存数据的容器（限制在 1MB 以内），也构成了一个层次化的命名空间。

ZooKeeper 目录中的每一个节点对应着一个 znode，每个 znode 维护着一个属性结构，它包含数据的版本号、时间戳等信息。ZooKeeper 就是通过这些属性来实现它的特定功能的。每当 znode 的数据改变时，相应的版本号就会增加；每当客户端查询、更新和删除数据时，也必须提供要被操作的 znode 版本号。如果提供的数据版本号与实际不匹配，那么操作将会失败。

znode 可以是临时节点，一旦创建这个 znode 的客户端与服务器失去联系，这个 znode 将自动删除。ZooKeeper 的客户端和服务器通信采用长连接方式，每个客户端和服务器通过心跳来保持连接，这种连接状态称为 session。如果 znode 是临时节点，则这个 session 失效，znode 也就被删除了。

znode 的节点属性如表 4-3 所示。

表 4-3 znode 的节点属性

属　性	描　述
czxid	节点被创建时的 zxid 值
mzxid	节点被修改时的 zxid 值
ctime	节点创建的时间
mtime	节点最后一次修改时间
vesion	节点的版本号
cversion	节点所拥有的子节点被修改的版本号
aversion	节点的 ACL 被修改的版本号
dataLength	节点数据的长度
numChildren	节点拥有子节点的个数
ephemeralOwner	如果节点为临时节点，那么它的值为这个节点拥有者的 session ID；否则它的值为 0

znode 是客户端访问 ZooKeeper 的主要实体，它包含以下主要特征。

1. watch

znode 可以被监控，包括这个目录节点中存储的数据的修改、子节点目录的变化等，一旦发生变化便可以通知设置监控的客户端。这是 ZooKeeper 的核心特性，ZooKeeper 的很多功能都是基于这个特性实现的。

znode 状态发生改变时（增、删、改等操作），watch（监视器）机制可以让客户端得到通知，并且仅仅会触发一次 watch。

在读操作 exists、getChildren 和 getData 上可以设置监视器，这些监视器可以被写操作 create、delete 和 setData 触发。ACL 相关的操作不参与任何监视器。当一个监视器被触发时，会产生一个事件，这个监视器和触发它的操作共同决定了监视器事件类型。

- 当所监视的 znode 被创建子节点、删除或更新时，设置在 exists 操作上的监视器将会被触发。
- 当所监视的 znode 被删除或更新时，设置在 getData 上的监视器将会被触发。创建 znode 不会触发 getData 上的监视器，因为 getData 操作成功执行的前提是 znode 已经存在。

- 当所监视的 znode 的一个子节点被创建或删除时，或监视的 znode 自己被删除时，设置在 getChildren 操作上的监视器将会被触发。

2. 数据访问控制

每个 znode 创建时都会有一张 ACL 列表，用于决定谁可以执行哪些操作。

3. 临时节点

ZooKeeper 节点有两种类型：临时节点和持久节点。节点类型在创建时确定，并且不能修改。临时节点的生命周期依赖于创建它的会话，一旦会话结束，临时节点将会被删除。临时节点不允许有子节点。

4. 顺序节点

当创建 znode 时设置了顺序节点，那么该 znode 路径之后便会附加一个递增的计数，这个计数对于此节点的父节点来说是唯一的。

例如，一个客户端请求创建一个名为 /a/b- 的顺序节点，则所创建 znode 的名字可能是 a/b-5；稍后另外一个名为 /a/b- 的顺序节点被创建，计数器会给出一个更大的值来保证 znode 名称的唯一性，例如 /a/b-6。

4.4.5 ZooKeeper 的常用接口及操作

ZooKeeper 作为一个分布式的服务框架，主要用来解决分布式集群中应用系统的一致性问题，它能提供基于类似文件系统的目录节点树方式的数据存储。但是 ZooKeeper 并不是专门用来存储数据的，它的主要作用是维护和监控存储的数据的状态变化。通过监控这些数据状态的变化，可以达到基于数据的集群管理。后面将会详细介绍 ZooKeeper 能够解决的一些典型问题，这里先介绍一下 ZooKeeper 的操作接口和简单使用示例。

1. 常用接口

客户端要连接 ZooKeeper 服务器，可以通过创建 org.apache.zookeeper. ZooKeeper 的一个实例对象，然后调用这个类提供的接口来和服务器交互。

前面说过 ZooKeeper 主要用来维护和监控一棵目录节点树中存储的数据的状态，所以 ZooKeeper 的常用操作和目录节点树的操作大体一致，如创建一个目录节点、给某个目录节点设置数据、获取某个目录节点的所有子目录节点、给某个

目录节点设置权限和监控这个目录节点的状态变化。

这些接口如表 4-4 所示。

表 4-4 org.apache.zookeeper. ZooKeeper 方法列表

方法名	方法功能描述
Stringcreate(String path, byte[] data, List <ACL> acl, CreateMode createMode)	创建一个给定的目录节点 path，并给它设置数据。CreateMode 标识有 4 种形式的目录节点，分别是：PERSISTENT，持久化目录节点，这个目录节点存储的数据不会丢失；PERSISTENT_SEQUENTIAL，顺序自动编号的目录节点，这种目录节点会根据当前已经存在的节点数自动加 1，然后返回给客户端已经成功创建的目录节点名；EPHEMERAL，临时目录节点，一旦创建这个节点的客户端与服务器端口也就是 session 超时，这种节点会被自动删除；EPHEMERAL_SEQUENTIAL，临时自动编号节点
Statexists(String path, boolean watcher)	判断某个 path 是否存在，并设置是否监控这个目录节点。这里的 watcher 是在创建 ZooKeeper 实例时指定的 watcher。该方法还有一个重载方法，可以指定特定的 watcher
Statexists(String path, Watcher watcher)	重载方法，这里给某个目录节点设置特定的 watcher。watcher 在 ZooKeeper 中是一个核心功能，可以监控目录节点数据及子目录的状态，一旦这些状态发生变化，服务器就会通知所有设置在这个目录节点上的 watcher，从而使每个客户端很快知道它所关注的目录节点的状态发生变化，进而做出相应的反应
void delete(String path, int version)	删除 path 对应的目录节点，version 为 –1 可以匹配任意版本，也就删除了这个目录节点的所有数据
List<String> getChildren(String path, boolean watch)	获取指定 path 下的所有子目录节点。同样，getChildren 方法也有一个重载方法，可以设置特定的 watcher 监控子节点的状态
StatsetData(String path, byte[] data, int version)	给 path 设置数据，可以指定这个数据的版本号，如果 version 为 –1 则可以匹配任意版本
byte[] getData(String path, boolean watch, Stat stat)	获取这个 path 对应的目录节点存储的数据，数据的版本等信息可以通过 stat 来指定，同时还可以设置是否监控这个目录节点数据的状态
void addAuthInfo(String scheme, byte[] auth)	客户端将自己的授权信息提交给服务器，服务器将根据这个授权信息验证客户端的访问权限

续表

方法名	方法功能描述
StatsetACL(String path, List<ACL> acl, int version)	给某个目录节点重新设置访问权限。需要注意的是，ZooKeeper 中的目录节点权限不具有传递性，父目录节点的权限不能传递给子目录节点。目录节点 ACL 由两部分组成：Perms 和 Id。Perms 有 ALL、READ、WRITE、CREATE、DELETE、ADMIN 几种；而 Id 标识了访问目录节点的身份列表，默认情况下有以下两种：ANYONE_ID_UNSAFE = new Id("world", "anyone") 和 AUTH_IDS = new Id("auth", "")，分别表示任何人都可以访问和创建者拥有访问权限
List<ACL> getACL(String path, Stat stat)	获取某个目录节点的访问权限列表

除了上述方法外，还有一些重载方法，如都提供了一个回调类的重载方法，以及可以设置特定 watcher 的重载方法。具体的方法可以参考 org.apache.zookeeper.ZooKeeper 类的 API 说明。

2. 基本操作

下面给出了 ZooKeeper 基本操作的示例代码，这样就能对 ZooKeeper 有一个直观的认识。下面的举例包括了创建与 ZooKeeper 服务器的连接，以及最基本的数据操作。

```
// 创建一个与服务器的连接
ZooKeeper zk = new ZooKeeper("localhost:" + CLIENT_PORT,
    ClientBase.CONNECTION_TIMEOUT, new Watcher() {
      // 监控所有被触发的事件
      public void process(WatchedEvent event) {
        System.out.println("已经触发了" + event.getType() + "事件！");
      }
    });
// 创建一个目录节点
zk.create("/testRootPath", "testRootData".getBytes(), Ids.OPEN_ACL_UNSAFE,
 CreateMode.PERSISTENT);
// 创建一个子目录节点
 zk.create("/testRootPath/testChildPathOne", "testChildDataOne".getBytes(),
  Ids.OPEN_ACL_UNSAFE,CreateMode.PERSISTENT);
System.out.println(new String(zk.getData("/testRootPath",false,null)));
// 取出子目录节点列表
System.out.println(zk.getChildren("/testRootPath",true));
// 修改子目录节点数据
```

```
 zk.setData("/testRootPath/testChildPathOne","modifyChildDataOne".
getBytes(),-1);
 System.out.println("目录节点状态：["+zk.exists("/testRootPath",true)+"]");
 //创建另外一个子目录节点
 zk.create("/testRootPath/testChildPathTwo", "testChildDataTwo".
getBytes(),
  Ids.OPEN_ACL_UNSAFE,CreateMode.PERSISTENT);
 System.out.println(new String(zk.getData("/testRootPath/testChildPathTwo",
true,null)));
 //删除子目录节点
 zk.delete("/testRootPath/testChildPathTwo",-1);
 zk.delete("/testRootPath/testChildPathOne",-1);
 //删除父目录节点
 zk.delete("/testRootPath",-1);
 //关闭连接
 zk.close();
```

输出结果如下：

```
已经触发了 None 事件!
 testRootData
 [testChildPathOne]
目录节点状态：[5,5,1281804532336,1281804532336,0,1,0,0,12,1,6]
已经触发了 NodeChildrenChanged 事件!
 testChildDataTwo
已经触发了 NodeDeleted 事件!
已经触发了 NodeDeleted 事件!
```

当目录节点的监控状态被打开时，一旦目录节点的状态发生变化，watcher 对象的 process() 方法就会被调用。

4.4.6 ZooKeeper 的应用场景分析

从设计模式角度来看，ZooKeeper 是一个基于观察者模式设计的分布式服务管理框架，它负责存储和管理大家都关心的数据，然后接受观察者的注册。一旦这些数据的状态发生变化，ZooKeeper 将负责通知已经在 ZooKeeper 上注册的那些观察者做出相应的反应，从而实现集群中类似 Master/Slave 的管理模式。

下面我们就来看看 ZooKeeper 适用于哪些应用场景，以及在这些场景中是如何解决问题的。

1. 统一命名服务（Name Service）

在分布式应用中，通常需要一套完整的命名规则，既能够产生唯一的名称，又便于识别和记住。通常情况下采用树形的名称结构是一个理想的选择。树形的名称结构是一个有层次的目录结构，既对人友好又不会重复。说到这里读者可能想到了 JNDI，没错，ZooKeeper 的 Name Service 与 JNDI 能够完成的功能是差不多的，它们都是将有层次的目录结构关联到一定资源上。但是 ZooKeeper 的 Name Service 是广泛意义上的关联，也许你并不需要将名称关联到特定资源上，只需要此名称不重复，就像数据库中产生一个唯一的数字主键一样。

Name Service 已经是 ZooKeeper 内置的功能，只需调用 ZooKeeper 的 API 就能实现。如调用 create 接口就可以很容易地创建一个目录节点。

2. 配置管理（Configuration Management）

配置管理在分布式应用环境中很常见，例如同一个应用系统需要多台 PC Server 运行，但是它们运行的应用系统的某些配置项是相同的，如果要修改这些相同的配置项，那么就必须同时修改每台运行这个应用系统的 PC Server，这样非常麻烦而且容易出错。

像这样的配置信息完全可以交给 ZooKeeper 来管理。将配置信息保存在 ZooKeeper 的某个目录节点中，然后将所有需要修改的应用机器监控配置信息的状态，一旦配置信息发生变化，每台应用机器就会收到 ZooKeeper 的通知，然后从 ZooKeeper 获取新的配置信息应用到系统中，如图 4-12 所示。

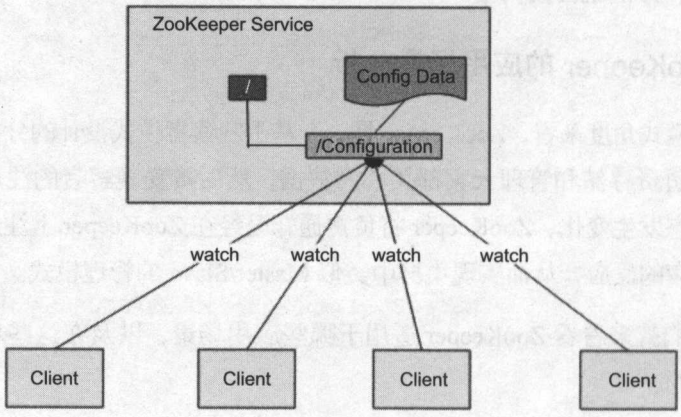


图 4-12　配置管理结构图

3. 集群管理（Group Membership）

ZooKeeper 能够很容易地实现集群管理功能，如有多台 Server 组成一个服务集群，那么必须要有一个“总管”知道当前集群中每台机器的服务状态，一旦有机器不能提供服务，集群中的其他机器必须知道，从而做出调整，重新分配服务策略。同样，当增加集群的服务能力时，就会增加一台或多台 Server，同样也必须让“总管”知道。

ZooKeeper 不仅能够维护当前集群中机器的服务状态，而且能够选出一个“总管”，让这个“总管”来管理集群，这就是 ZooKeeper 的另一个功能—— Leader Election。

它们的实现方式都是在 ZooKeeper 上创建一个 EPHEMERAL 类型的目录节点，然后每个 Server 在它们创建目录节点的父目录节点上调用 getChildren(String path, boolean watch) 方法并设置 watch 为 true。由于是 EPHEMERAL 目录节点，当创建它的 Server 出现故障时，这个目录节点也随之被删除，所以 Children 将会变化，这时 getChildren 上的 Watch 将会被调用，所以其他 Server 就会知道某台 Server 出现了故障。新增 Server 也是同样的原理。

ZooKeeper 实现 Leader Election，也就是选出一个 Master Server。和前面一样，每台 Server 创建一个 EPHEMERAL 目录节点，不同的是它还是一个 SEQUENTIAL 目录节点，所以它是一个 EPHEMERAL_SEQUENTIAL 目录节点。之所以它是 EPHEMERAL_SEQUENTIAL 目录节点，是因为我们可以给每台 Server 编号，我们可以选择当前是最小编号的 Server 为 Master。假如这个最小编号的 Server 出现故障，由于是 EPHEMERAL 节点，出现故障的 Server 对应的节点也被删除，所以当前的节点列表中又出现一个最小编号的节点，我们就选择这个节点为当前 Master。这样就实现了动态选择 Master，避免了传统意义上单 Master 容易出现单点故障的问题。

4. 共享锁（Locks）

共享锁在同一个进程中很容易实现，但是在跨进程或者在不同 Server 之间就不易实现了。ZooKeeper 可以很容易地实现这个功能，实现方式也是需要获得锁的 Server 创建一个 EPHEMERAL_SEQUENTIAL 目录节点，然后调用 getChildren() 方法获取当前的目录节点列表中最小的目录节点，看是不是自己创建的目录节

点。如果正是自己创建的，那么它就获得了这个锁；如果不是，那么它就调用 exists(String path, boolean watch) 方法并监控 ZooKeeper 上目录节点列表的变化，一直到自己创建的节点是列表中最小编号的目录节点，从而获得锁。释放锁很简单，只要删除前面它自己创建的目录节点即可。

5. 队列管理

当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达，这种队列是同步队列。队列按照 FIFO 方式进行入队和出队操作，例如实现生产者和消费者模型。

同步队列用 ZooKeeper 实现的思路为：创建一个父目录 /synchronizing，每个成员都监控标志（Set Watch）位目录 /synchronizing/start 是否存在，接着每个成员都加入这个队列。加入队列的方式就是创建 /synchronizing/member_i 的临时目录节点，然后每个成员获取 /synchronizing 目录的所有目录节点，也就是 member_i。最后判断 i 的值是否已经是成员的个数，如果小于成员个数，则等待 /synchronizing/start 的出现；如果已经相等，则创建 /synchronizing/start。

4.5 小结

本章所述的 Hadoop“四剑客”在整个 Hadoop 阵营中起着中流砥柱的作用。

Hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供简单的 SQL 查询功能，可以将 SQL 语句转换为 MapReduce 任务来运行。其优点是学习成本低，可以通过类 SQL 语句快速实现简单的 MapReduce 统计，不必开发专门的 MapReduce 应用，十分适合数据仓库的统计分析。

HBase 是一个分布式的、面向列的开源数据库。就像 Bigtable 利用了 Google 文件系统所提供的分布式数据存储一样，HBase 在 Hadoop 之上提供了类似 Bigtable 的能力。HBase 是 Apache 的 Hadoop 项目的子项目。HBase 不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库。另一个不同点是 HBase 是基于列的而不是基于行的模式。

Pig 是一种探索大规模数据集的脚本语言。MapReducer 的一个主要缺点就是开发周期太长。Pig 的强大之处就是它只要几行 Pig Latin 代码就能处理 TB 级别的

数据。Pig提供了多个命令用于检查和处理程序中的数据结构，因此它能很好地支持写查询。Pig的一个很有用的特性就是它支持在输入数据中有代表性的一个小的数据集上试运行。

ZooKeeper是一个开放源码的分布式应用程序协调服务，是Google的Chubby的一个开源实现，是Hadoop和HBase的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括配置维护、域名服务、分布式同步、组服务等。

5 Hadoop 资源管理与调度

5.1 Hadoop 调度机制

通常 Hadoop 集群是作为一个整体对外服务的，考虑的是集群整体资源利用的最大化，HDFS 的多副本策略使得集群中的某一单独节点运行是否正常不再重要。Hadoop 的一个重要理念是移动计算而不是数据，所以 Hadoop 会尽量在数据存储的节点进行运算。接下来我们看看 Hadoop 的内置调度机制。

在 2008 年以前，集成在 JobTracker 中的调度算法仅有 FIFO。在 FIFO 调度中，JobTracker 从工作队列中拉取作业，先进先出。当 Hadoop 最初出现的时候，主要设计用来进行大型的批处理，例如网页索引、日志挖掘，用户将任务提交至队列，而后集群按照提交的先后顺序执行即可，FIFO 也足够用了。然而，当越来越多的数据放在 Hadoop 集群后，另一个问题出现了，人们开始希望在多个用户间共享集群。而此时 Hadoop 只支持与 JobTracker 逻辑混合在一起的单一调度器。

于是 2008 年 5 月 18 日有人提交了一个有建设性的提议：HADOOP-3412（Refactor the scheduler out of the JobTracker），调度器从 JobTracker 的代码中分离出来，变成可插入式的，在 JobTracker 中加载和调用。用户可以在配置文件 mapred-site.xml 的 mapred.jobtracker.taskScheduler 属性中指定自己的调度器。

当前 Hadoop 自带的调度器随着 Hadoop 版本的不同会略有区别，但是最常见的三种调度器分别是 FIFO（默认调度器）、Capacity Scheduler（计算能力调度器）

和 Fair Scheduler（公平调度器）。以下逐一介绍其算法及适用场景。

5.1.1 FIFO

FIFO（First In First Out）是 Hadoop 最初的调度器。FIFO 比较简单，Hadoop 中只有一个作业队列，被提交的作业按照先后顺序在作业队列中排队，新来的作业插入到队尾。一个作业运行完成后，JobTracker 总是从队首取下一个作业运行。这种调度策略的优点是简单、易于实现，同时也减轻了 JobTracker 的负担。但是它的缺点也是显而易见的，它对所有的作业一视同仁，没有考虑到作业的紧迫程度，而且对小作业的运行不利。

在 Hadoop 的后期版本中，增加了优先级处理功能，可以通过设置 mapred.job.priority 属性或者在 JobClient 中调用 setJobPriority() 方法来进行优先级设置（优先级可以设置为 VERY_HIGH、HIGH、NORMAL、LOW、VERY_LOW）。Job Scheduler 在选择下一个运行作业时，会选择一个最高优先级的作业，而后再根据提交时间排序。但是由于 FIFO 调度器不支持抢占，所以一个高优先级作业仍然可能被一个长期运行的低优先级作业所阻塞。

显然，这种调度机制很难满足各种场合的应用，尤其是有些场合对业务的优先级看得很重，进而发展出了其他的调度方式。

5.1.2 计算能力调度器

计算能力调度器（Capacity Scheduler）是雅虎开发并贡献的，其设计目标是让 Hadoop 应用程序真正成为一个共享的、多租户的集群，并且使得集群的吞吐量和利用率最大化。说得有点绕口，下面详细说明计算能力调度器的来龙去脉。

在传统企业中，每个部门或者每个项目都有自己的“小金库”（私有服务器），用这些私有服务器来达到服务的质量和处理的速度，但问题是，这样就会造成资源的浪费，资源的平均利用率不高。当用到 Hadoop 集群的时候，就需要各个团队把他们的“小金库”共享出来，进而提高资源利用率。但问题来了，如果把资源共享出来，当我们急切需要资源运行高优先级的任务的时候，分配不到资源怎么办呢？为了解决这个问题，计算能力调度器便产生了。

计算能力调度器正是设计用来共享一个大规模的集群，并给予每个系统性能保证。其中心思想是各系统按照自己计算的需要，共同投资一个 Hadoop 集群，集

群中的可用资源在各系统中分享。这样的集群有一个附加的好处，即单个系统可以利用其他系统的闲置计算资源进行计算，使得资源的利用更加富有弹性及高效。

而在各系统间共享集群就必须使系统对多租户（multi-tenancy）设计有很强的支持，因为必须给予每个系统性能保证，并且要确保集群在遇到一个异常的应用 / 用户 / 数据集等时也不受影响。计算能力调度器提供一系列严格的限制来确保一个应用 / 用户 / 队列不能消耗不恰当的集群资源，同时还限制了一个单一的用户 / 队列可以初始化 / 等待的应用程序数量来确保集群的公平使用和稳定性。

计算能力调度是以队列为单位分配 TaskTracker（集群中的一个节点）的，这种调度策略配置了多个队列，每个队列配置了最小额度的 TaskTracker 数量，当一个队列有空闲的 TaskTracker 时，调度器会将空闲的分配给其他队列。由于这时候可能有多个队列没有得到最小额度的 TaskTracker 而又在申请新的，空闲的 TaskTracker 会被优先分配到最饥饿的队列中去。如何衡量饥饿程度呢？可以通过计算队列中正在运行的任务数与其分得的计算资源之间的比值是否最低来判断，比值越低说明饥饿程度越高。

计算能力调度策略是以队列的方式组织作业的，所以一个用户的作业可能在多个队列中，如果不对用户进行一定的限制，则很可能出现在多个用户之间严重不公平的现象。所以在选中新作业运行时，还需要考虑作业所属的用户是否超过了资源的限制，如果超过，则作业不会被选中。

对于同一个队列中的作业，这种策略是基于优先级的 FIFO 策略，但是不会抢占资源。

计算能力调度器的特性有：

- 支持多用户多队列，支持多系统多用户共享集群。
- 单个队列是 FIFO 调度方式，队列内可以开启优先级控制功能（优先级控制默认处于关闭状态）。
- 支持资源共享，当队列内的资源有剩余时，可共享给其他缺少资源的队列。
- 当某个 TaskTracker 上出现空闲 slot 时，调度器的调度策略是：先选择资源利用率（numSlotsOccupied/capacity）低的队列，然后在队列中考虑 FIFO 和优先级进行作业选择，选择的时候会判断作业所在的用户资源未超限，以及有足够内存满足作业的内存需求。

- 计算能力调度器在调度 Job 时会考虑作业的内存限制，为了满足某些特殊 Job 的特殊内存需求，可能会为该 Job 分配多个 slot。例如一个 slot 占用 1GB 内存，一个需要 2GB 内存的程序就会被分配两个 slot。FIFO 及公平调度器不具备此功能。

5.1.3 公平调度器

公平调度器是由 Facebook 贡献的，适合于多用户共享集群环境。公平调度器的设计目标是即使在与大作业共享集群的时候也能迅速完成小作业。与 Hadoop 默认的 FIFO 调度器不同的是，公平调度器可以在大作业运行时让小作业的执行也能取得进展，同时又不会使大作业处于资源“饥饿”状态；支持在运行时进行重配置，而不需要重启集群；便于管理及配置，程序在遇到一些异常情况的时候也能做出合适的处理，用户也仅在他们需要一些高级功能的时候才需要进行配置。

公平调度器按资源池（Pool）来组织作业，并把资源公平地分到这些资源池里。在默认情况下，每个用户拥有一个独立的资源池，以使每个用户都能获得一份等同的集群资源而不管他们提交了多少作业。按用户的 UNIX 群组或作业配置（jobconf）属性来设置作业的资源池也是可以的。在每一个资源池内，会使用公平共享（Fair Sharing）的方法在运行作业之间共享容量（Capacity）。用户也可以给予资源池相应的权重，以不按比例的方式共享集群。

除了提供公平共享的方法外，公平调度器允许赋予资源池保证（guaranteed）最小共享资源，这在确保特定用户、群组或生产应用程序总能获取到足够的资源时是很有用的。当一个资源池包含作业时，它至少能获取到它的最小共享资源；但是当资源池不完全需要它所拥有的保证共享资源时，额外部分会在其他资源池间进行切分。

这种策略在系统中配置了 slot，一个 slot 可以运行一个任务，这些任务就是一个大的作业被切分后的小作业。当一个用户提交多个作业时，每个作业可以分配到一定的 slot 以执行任务。如果把整个 Hadoop 集群作业调度与操作系统的作业调度相比，FIFO 就相当于操作系统中早期的单道批处理系统，系统中每个时刻只有一道作业在运行；而公平调度器相当于多道批处理系统，它实现了同一时刻多道作业同时运行。由于 Linux 是多用户的，若有多个用户同时提交多个作业会怎样？在这种策略中给每个用户分配一个作业池，然后给每个作业池设置一个最小共享

slot 个数。什么是最小共享 slot 个数呢？最小是指只要这个作业池需要，调度器应该确保能够满足这个作业池的最小 slot 个数的需求。但是如何才能确保在它需要的时候就有空的 slot？一种方法是固定分配一定数量的 slot 给作业池，这个数量至少是最小 slot 值，这样只要在作业池需要的时候分配给它就可以了，但是这样在这个作业池没有用到这么多 slot 的时候就会造成浪费。这种策略实际上是这样做的：当作业池的需求没有达到最小 slot 个数时，名义上是自己剩余的 slot 会被分配给其他有需要的作业池，当一个作业池需要申请 slot 的时候，若系统中没有了，也不会去抢占别人的，只要当前一个空的 slot 释放就会被立即分配给这个作业池。

在一个用户的作业池内，这种调度策略分为两级：

第一级，在池间分配槽，在多用户的情况下，每个用户分配一个作业池。

第二级，在作业池内，每个用户可以使用不同的调度策略。

公平调度器的特性有：支持多用户多队列；资源公平共享（公平共享量由优先级决定）；保证最小共享量；支持时间片抢占；限制作业并发量，以防止中间数据塞满磁盘。

5.2 Hadoop YARN 资源调度

5.2.1 YARN 产生的背景

什么是 YARN? 其实它是几个英文单词的缩写：Yet Another Resource Negotiator。如果强行翻译的话，是“另一种资源协调者”的意思。有读者就会想，这既然是另一种，那前一种是什么？前一种其实就是之前我们提到的 Hadoop MapReduce 中的调度方法。YARN 出现后，有人将其称为 MapReduce V2，其实这种叫法不是很严谨，但至少从这种叫法上我们可以看出 YARN 的引入使得 Hadoop 有了很大的提升，具有质的突破。图 5-1 所示为 Hadoop 从 1.0 到 2.0 的进化图。最主要的变化是在 2.0 版本中增加了 YARN。

从本质上来说，YARN 是 Hadoop 的操作系统，突破了 MapReduce 框架的性能瓶颈。同时它也是一个真正的 Hadoop 资源管理器，允许多个应用程序同时、高效地运行在一个集群上。有了 YARN，Hadoop 将是一个真正的多应用程序平台，可服务于整个企业，并解决了 1.0 版本的一些难题。那么都解决了哪些主要问题呢？

从宏观上看，主要解决的问题有两个：扩展性和主节点单点故障。如果究其枝末的话，在性能、多租户等问题上也有所提升。接下来我们看看具体的痛点在哪里。

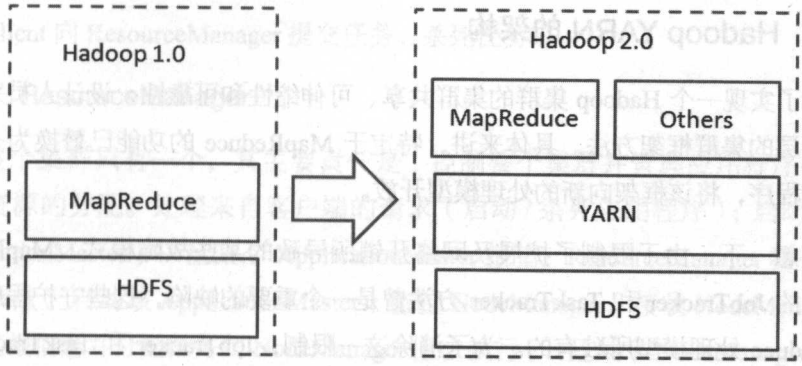


图 5-1　Hadoop 从 1.0 到 2.0 的进化图

问题一：在使用 MapReduce 第一个版本的时候，当集群包含的节点超过 4000 个时（其中每个节点可能是多核的），就会表现出一定的不可预测性。其中一个最大的问题是级联故障。由于要尝试复制数据和重载活动的节点，所以一个故障会通过网络泛洪形式导致整个集群严重恶化。

问题二：在 HDFS 中，其主要文件系统元数据的管理都是通过 NameNode（主节点）完成的，真实的数据文件被切割成大小为 64MB 的块（Block），分散放在所有 DataNode（从节点）上，并且保证每个块都会被复制三份，而且这三份不能放在同一节点上。当任意一个 DataNode 发生故障时，NameNode 可从其他从节点恢复故障从节点上的数据，并将数据复制到其他可用的从节点上去。可一旦 NameNode 发生故障，在默认的机制下，则没有办法自动恢复 NameNode 上的元数据信息，只能靠人工的方式通过元数据信息的备份来重建 NameNode，然后再将整个 HDFS 的集群运行起来。很多人认为这样的工作机制是有问题的，因为不会有任何一个场景可以容忍系统中断而无容错机制，尤其是在设计时就不加以考虑。这看起来确实不可思议，但如果我们对 Hadoop 的发展及版本进化过程有所了解，就会知道，其实在 Hadoop 最开始的应用场景中，确实没有高可靠性和扩展性等其他方面特点的考虑，或者说，Hadoop 在开始的时候根本不是为了解决我们现在所面临的这些问题的，只是在进化的过程中需要面对和解决的问题越来越多，从设计上既要兼容原有的代码，又要能够对未来有所预见，所以才逐渐解决了我们现在所说的单点故障问题。

那么，这两个问题在 YARN 中是如何解决的呢？请看下面介绍的 YARN 的架构，掌握了其架构，问题自然得解。

5.2.2 Hadoop YARN 的架构

为了实现一个 Hadoop 集群的集群共享、可伸缩性和可靠性，设计人员采用了一种分层的集群框架方法。具体来讲，特定于 MapReduce 的功能已替换为一组新的守护程序，将该框架向新的处理模型开放。

回想一下，由于限制了扩展及网络开销所导致的某些故障模式，MapReduce v1.0 中的 JobTracker 和 TaskTracker 方法曾是一个重要的缺陷，这些守护程序也是 MapReduce 处理模型所独有的。为了消除这一限制，JobTracker 和 TaskTracker 已从 YARN 中删除，取而代之的是一组对应用程序不可知的新守护程序。

YARN 采用了 Master/Slave 结构，一个 ResourceManager 对应多个 NodeManager。YARN 由 Client、ResourceManager、NodeManager、ApplicationMaster（图中的 App Mstr）、Container 组成，如图 5-2 所示。

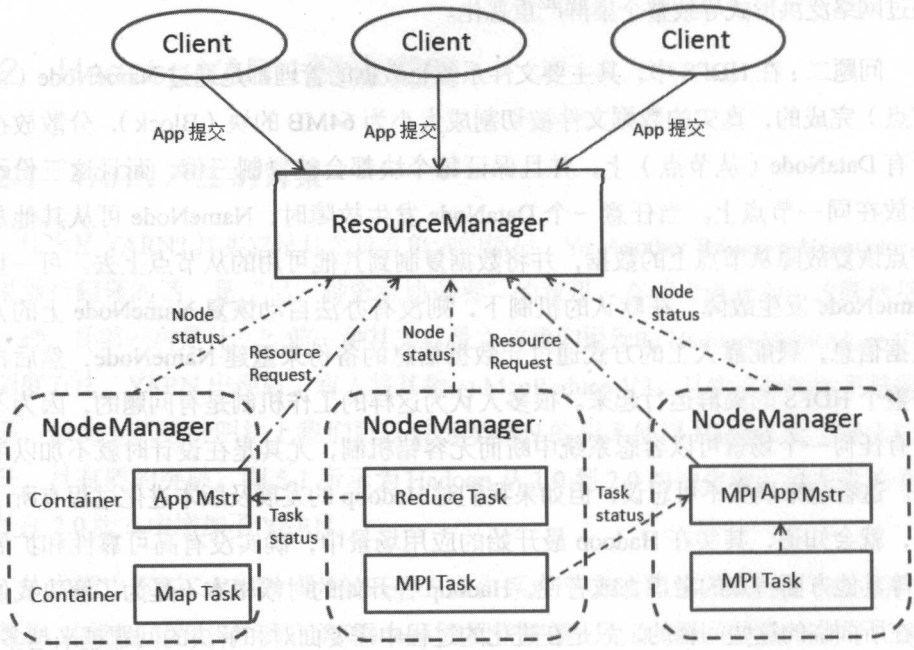


图 5-2 YARN 的运行架构

接下来我们从各个模块入手，来分析 YARN 是怎样运转起来的。

1. Client

Client 向 ResourceManager 提交任务、杀死任务等。

2. ResourceManager

整个集群只有一个，其主要责任是：控制整个集群并管理应用程序向基础计算资源的分配。处理来自客户端的请求（启动 / 杀死应用程序）；启动 / 监控 ApplicationMaster；一旦某个 ApplicationMaster 报错，ResourceManager 将会在另一个节点上启动该 ApplicationMaster；监控 NodeManager，接收 NodeManager 的心跳汇报信息并分配任务到 NodeManager 上去执行；一旦某个 NodeManager 报错，标识一下该 NodeManager 上的任务，来告诉对应的 ApplicationMaster 如何处理；负责整个集群的资源分配和调度，将各个资源部分（计算、内存、带宽等）精心安排给基础 NodeManager。从某个方面来讲，ResourceManager 承担了 JobTracker 的角色。

3. ApplicationMaster

每个应用一个，其主要责任是：管理一个在 YARN 内运行的应用程序的每个实例。负责应用程序的管理；数据切分；为应用程序 / 作业向 ResourceManager 申请资源（Container），负责协调来自 ResourceManager 的资源，并通过 NodeManager 监视容器的执行和资源使用（CPU、内存等的资源分配）；与 NodeManager 通信以启动 / 停止任务；任务监控和容错（在任务执行失败时重新为该任务申请资源以重启任务）；处理 ResourceManager 发过来的命令，如杀死 Container、让 NodeManager 重启等。

4. NodeManager

整个集群中有多个，其主要责任是：管理一个 YARN 集群中的每个节点。NodeManager 提供针对集群中每个节点的服务，从监督对一个 Container 的终生管理到监视资源和跟踪节点健康，周期性地向 ResourceManager 汇报本节点上的资源使用情况和各个 Container 的运行状态；接收并处理来自 ResourceManager 的 Container 启动 / 停止的各种命令；处理来自 ApplicationMaster 的命令；负责单个节点上的资源管理和任务调度。

5. Container

Container 是对任务运行环境的抽象，其主要责任是：任务运行资源（节点、内存、CPU）；任务启动命令；任务运行环境。任务是运行在 Container 中的，一个 Container 中既可以运行 ApplicationMaster，也可以运行具体的 Map/Reduce/MPI/Spark Task。

要使用一个 YARN 集群，首先需要来自包含一个应用程序的客户的请求。ResourceManager 协商一个 Container 的必要资源，启动一个 ApplicationMaster 来表示已提交的应用程序。通过使用一个资源请求协议，ApplicationMaster 协商每个节点上供应用程序使用的资源 Container。在执行应用程序时，ApplicationMaster 监视 Container 直到完成。当应用程序完成时，ApplicationMaster 从 ResourceManager 处注销其 Container，执行周期完成。

从 YARN 的架构中我们可以看出，YARN 很容易地解决了扩展性问题。那么关于容错性呢？ YARN 在容错性上也从以下几个方面下了功夫：

首先，关于 ResourceMananger，基于 ZooKeeper 实现了高可用性（High Availability，HA）避免单点故障。在 HA 环境中会存在一个 Active 节点和一个 Standby 节点，当 Active 节点出现问题时，Standby 节点可以转换为 Active 状态，从而让集群继续工作。这里的 HA 的实现和传统的方式在设计理念上有很多不一样的地方，具体会体现在 ZooKeeper 的服务上。

其次，对于 NodeManager，如果执行失败，则 ResourceManager 会将失败任务告诉对应的 ApplicationMaster，由 ApplicationMaster 决定如何处理。

最后，对于 ApplicationMaster，如果执行失败，则由 ResourceManager 负责重启，ApplicationMaster 会保存运行完成的任务，重启后无须重新运行。

5.2.3 YARN 的运作流程

YARN 的主要运作流程如图 5-3 所示。

YARN 的主要运作流程大致可分为以下几个阶段。

1. Job 提交阶段

用户向 YARN 中提交应用程序 / 作业，其中包括 ApplicaitonMaster 程序、

启动 ApplicationMaster 的命令、用户程序等；从 ResourceManager 中获取一个 Application ID 检查作业输出配置，计算输入分片，复制作业资源（job jar、配置文件、分片信息）到 HDFS，以便后面任务的执行。

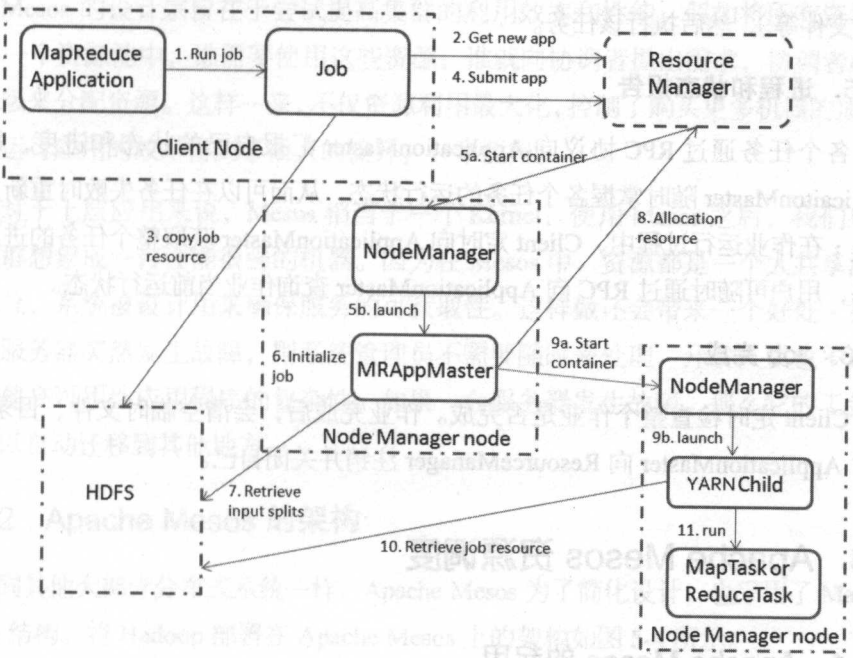


图 5-3 YARN 的运作流程

2. Job 初始化阶段

ResourceManager 将作业递交给 Scheduler（有很多调度算法，一般是根据优先级），Scheduler 为作业分配一个 Container，ResourceManager 就加载一个 ApplicationMaster Process 并交给 NodeManager 管理。ApplicationMaster 主要用来创建一系列的监控进程来跟踪作业的进度，同时获取输入分片，为每个分片创建一个 Map Task 和相应的 Reduce Task。ApplicationMaster 还决定如何运行作业，如果作业很小（可配置），则直接在同一个 JVM 下运行。

3. Task 分配阶段

ApplicationMaster 采用轮询的方式通过 RPC 请求向 ResourceManager 申请资源（一个个的 Container，指定任务分配的资源要求）。一般是根据 Data Locality 来分配资源的。

4. Task 执行阶段

一旦 ApplicationMaster 申请到资源后，它将根据 ResourceManager 的分配情况，在对应的 NodeManager 中启动 Container 从 HDFS 中读取任务所需资源（job jar、配置文件等），然后执行该任务。

5. 进程和状态报告

各个任务通过 RPC 协议向 ApplicationMaster 汇报自己的状态和进度，以便 ApplicaitonMaster 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；在作业运行过程中，Client 定时向 ApplicationMaster 获取整个任务的进度和状态，用户可随时通过 RPC 向 ApplicationMaster 查询作业当前运行状态。

6. Job 完成

Client 定时检查整个作业是否完成。作业完成后，会清空临时文件、目录等，同时 ApplicationMaster 向 ResourceManager 注销并关闭自己。

5.3 Apache Mesos 资源调度

5.3.1 Apache Mesos 的起因

Apache Mesos 是一款开源集群管理软件。从标题来看，Mesos 属于 Apache 基金会，但是 Mesos 却不是在 Apache 旗下诞生的。Mesos 是一个早在 2009 年由 Benjamin Hindman 等人联合在加利福尼亚大学伯克利分校的 AMPLab 发起的研究项目。Benjamin 随后将其引入 Twitter 公司，而如今它已经完美地运行在 Twitter 的数据中心上。由于它是开源的，参与开发的人越来越多，很多 IT 巨头都投入资源进行研究和应用。值得特别说明的是，Mesos 是一个资源调度框架，并非一整套完整的应用管理平台，但是它可以比较容易地跟各种应用管理或者中间件平台整合，提高资源使用效率。目前支持的上层应用也有很多，除了我们将要提到的 Hadoop 应用，还有 Spark、Storm、ElasticSearch、Kafka 等架构。

那么 Mesos 的初衷在于解决什么问题呢？在传统的实验室中，每个部门或者项目都有自己的机器，每台机器都有自己的职责。例如，我们将集群 A 分配给 Team-A 来执行 Spark 的应用，将集群 B 分配给 Team-B 来执行 Hadoop 的应用。当时，集群

A 和集群 B 可能不是并行运行的，即使并行运行，二者之间满负荷的时间段也可能是不一致的。假如说，9 点到 11 点，机群 A 已经超负荷工作，而集群 B 可能没有工作，或者负荷率很低，Team-A 用不到集群 B 的资源。Mesos 正是为了解决这一问题。

Mesos 的设计宗旨在于尝试提高集群的利用效率和性能。假如将所有资源都放到一个资源池中，谁需要使用这些资源，谁就向协调者提出需求，协调者根据优先级来分配资源。这样一来，不仅资源利用最大化，控制了购买更多机器的成本，而且运行应用的效率得到了很大的提升，一举两得。

对于上层应用来说，Mesos 相当于一个 Kernel，使用 Mesos 之后，我们可以将集群想象成一台性能很强的机器。因为在 Mesos 中，资源都是一个大共享池的一部分，系统被设计用来确保服务的可获取性。这样做还会带来一个好处：如果某台服务器突然发生故障，则系统管理员不需要随时来处理，开发人员也不必关心构建高可用性应用程序的复杂性。如果一台服务器发生故障，那么它的工作负载可以自动迁移到其他地方。

5.3.2 Apache Mesos 的架构

同其他大部分分布式系统一样，Apache Mesos 为了简化设计，也采用了 Master/Slave 结构。将 Hadoop 部署在 Apache Mesos 上的架构如图 5-4 所示。

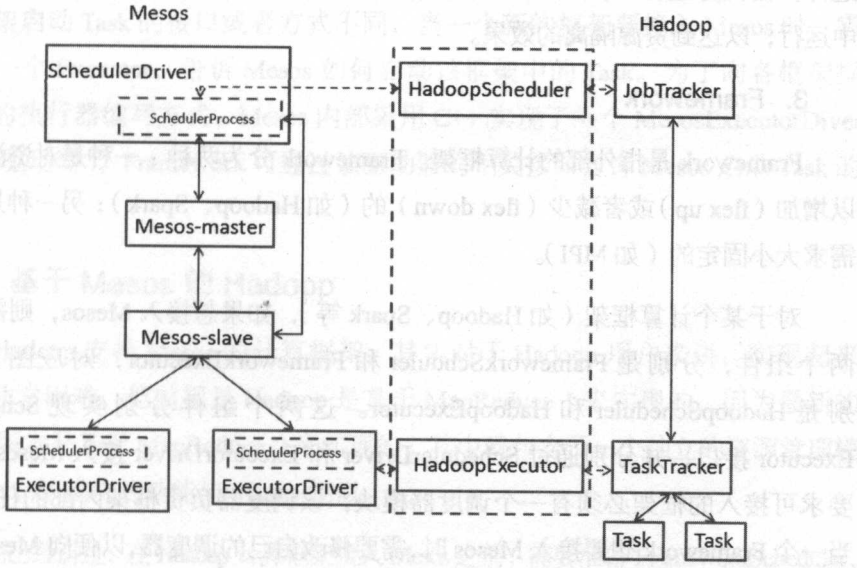


图 5-4 Hadoop 部署在 Apache Mesos 上的架构

Apache Mesos 由 5 个组件组成，分别是 Mesos-master、Mesos-slave、Framework、SchedulerProcess 和 ExecutorProcess。

1. Mesos–master

Mesos-master 是整个系统的核心，主要负责整体的资源调度和逻辑控制，管理接入 Mesos 的各个 Framework（由 frameworks_manager 管理）和 Slave（由 slaves_manager 管理），并将 Slave 上的资源按照某种策略分配给 Framework（由独立插拔模块 Allocator 管理）。很明显，Master 存在单点故障问题，为此，Mesos 采用 ZooKeeper 解决该问题。为了解决 Master 单点故障，将 Master 做得尽可能地轻量级，其上面所有的元数据都可以通过各个 Slave 重新注册而进行重构，故可以很容易地通过 ZooKeeper 解决该单点故障问题。

2. Mesos–slave

Mesos-slave 负责接收并执行来自 Mesos-master 的命令，管理节点上的 Mesos-task，并为各个 Task 分配资源。Mesos-slave 将自己的资源发送给 Mesos-master，由 Mesos-master 中的 Allocator 模块决定将资源分配给哪个 Framework。当前考虑的资源有 CPU 和内存两种，也就是说，Mesos-slave 会将 CPU 个数和内存量发送给 Mesos-master，而用户提交作业时，需要指定每个任务需要的 CPU 个数和内存量，这样，当任务运行时，Mesos-slave 会将任务放到包含固定资源的 Linux Container 中运行，以达到资源隔离的效果。

3. Framework

Framework 是指外部的计算框架。Framework 分为两种：一种是对资源需求可以增加（flex up）或者减少（flex down）的（如 Hadoop、Spark）；另一种是对资源需求大小固定的（如 MPI）。

对于某个计算框架（如 Hadoop、Spark 等），如果想接入 Mesos，则需要编写两个组件，分别是 FrameworkSchduler 和 FrameworkExecutor，对应图 5-4 中分别是 HadoopScheduler 和 HadoopExecutor。这两个组件分别实现 Scheduler 和 Executor 接口，并分别通过 SchedulerDriver 和 ExecutorDriver 接入 Mesos。Mesos 要求可接入的框架必须有一个调度器模块，该调度器负责框架内部的任务调度。当一个 Framework 想要接入 Mesos 时，需要修改自己的调度器，以便向 Mesos 注册，

并获取 Mesos 分配给自己的资源，这样再由自己的调度器将这些资源分配给框架中的任务。也就是说，整个 Mesos 系统采用双层调度框架：第一层，由 Mesos 将资源分配给框架；第二层，框架自己的调度器将资源分配给自己内部的任务。当前 Mesos 支持多种语言编写的调度器，如 C++、Java、Go、Python、Scala 等。

4. SchedulerProcess

为了向各种调度器提供统一的接入方式，Mesos 内部采用 C++ 实现了一个 SchedulerDriver（调度器驱动器），Framework 的调度器可调用该 Driver 中的接口与 Mesos-master 交互，完成一系列功能（如注册、资源分配等）。

Mesos-master 为各个 Framework 分配资源，这些资源直接传递给 Mesos SchedulerProcess，再由 SchedulerProcess 调用 FrameworkScheduler 的相关函数，由 FrameworkScheduler 将这些资源分配给框架中的任务，并返回给 Mesos-master，由 Mesos-master 转发给相应的 Mesos-slave。

如果 SchedulerProcess 中保存了 Mesos-slave 的地址，则直接将相关消息发送给 Mesos-slave，不必再由 Mesos-master 转发。

5. ExecutorProcess

ExecutorProcess 是实际工作的组件，主要用于启动框架内部的 Task。由于不同的框架启动 Task 的接口或者方式不同，当一个新的框架要接入 Mesos 时，需要编写一个 Executor，告诉 Mesos 如何启动该框架中的 Task。为了向各框架提供统一的执行器编写方式，Mesos 内部采用 C++ 实现了一个 MesosExecutorDiver（执行器驱动器），Framework 可通过该驱动器的相关接口告诉 Mesos 启动 Task 的方法。

5.3.3 基于 Mesos 的 Hadoop

让 Hadoop 支持 Mesos 的计算框架，其实对于 Hadoop 用户来讲，实现起来并不是非常困难，但前提是 Hadoop 是基于 MapReduce 1 来实现的，因为最新的 Hadoop 版本是基于 MapReduce 2 来实现的，其中就包含了一个独立的资源管理模块 YARN，该模块的功能与 Mesos 相似。

前面已经讲过，在 Hadoop 计算框架接入 Mesos 之前，需要准备 FrameworkScheduler，

由它来调度从 Mesos-master 处获取的资源，并且寻找与资源相匹配的任务，通过 MesosSchedulerDriver 来和 Mesos-master 进行交互。

关于 Hadoop 和 Mesos 的安装，笔者不想赘述，读者可以参照官方的指导，笔者之意在于把流程从设计思想的角度进行分析讨论。

首先来看看基于 Mesos 的 Hadoop 的 mapred-site.xml。

```
<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>master:54311</value>
  </property>
  <property>
    <name>mapred.jobtracker.taskScheduler</name>
    <value>org.apache.hadoop.mapred.MesosScheduler</value>
  </property>
  <property>
    <name>mapred.mesos.master</name>
    <value>master:6060</value>
  </property>
</configuration>
```

通过 mapred.jobtracker.taskScheduler 这个属性的值，将 Mesos 的调度作为一个插件，替换掉原来的调度器。这里的调度其实就是我们前面讲到的计算框架接入 Mesos 之前实现的 FrameworkScheduler，它是 MesosScheduler 的一个子类。

在 JobTracker 中有如下代码，从这段代码中我们可以看到，JobTracker 用于从配置文件中读取 mapred.jobtracker.taskScheduler 这个属性的值，即 org.apache.hadoop.mapred.MesosScheduler。

```
Class<? extends TaskScheduler> schedulerClass
  = conf.getClass("mapred.jobtracker.taskScheduler",
      JobQueueTaskScheduler.class, TaskScheduler.class);
  taskScheduler= (TaskScheduler) ReflectionUtils.newInstance(schedulerClass,
conf);
```

读到整个值以后，Hadoop 就可以判断资源调度平台不再是默认值，而是采用热插拔的形式替换为了 Mesos。接下来就进入 Mesos的注册过程。注册过程如图 5-5 所示。

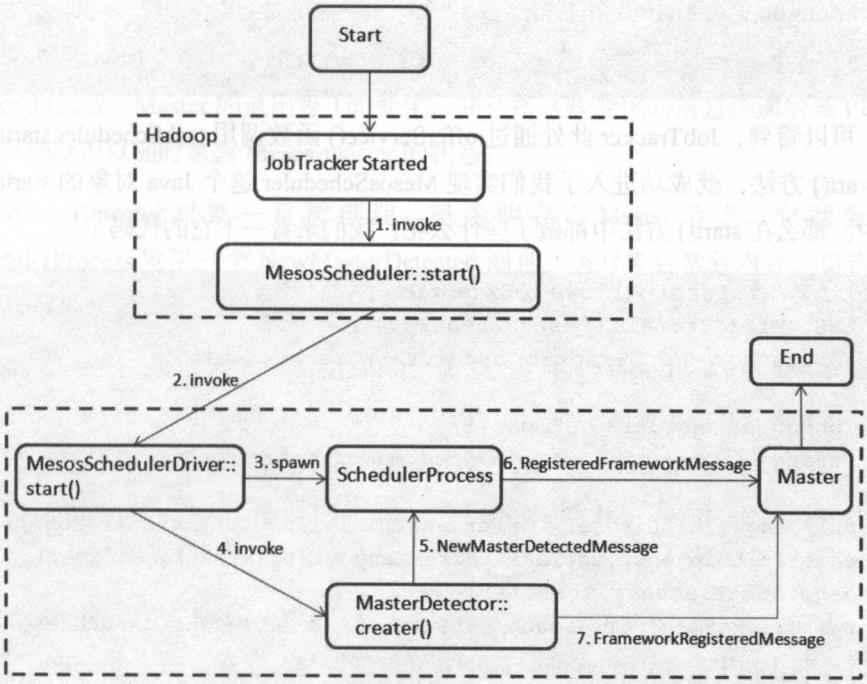


图 5-5 Mesos 的注册过程

注册过程大致如下。

JobTracker 启动时，会调用 MesosScheduler 的 start() 方法。在 JobTracker 中有如下函数（省略了部分）：

```
public void offerService() throws InterruptedException, IOException {
    // Prepare for recovery. This is done irrespective of the status of tart
    // flag.
    while (true) {
      try {
        recoveryManager.updateRestartCount();
        break;
      } catch (IOException ioe) {
        LOG.warn("Failed to initialize recovery manager.", ioe);
        // wait for some time
        Thread.sleep(FS_ACCESS_RETRY_PERIOD);
        LOG.warn("Retrying...");
      }
    }
```

```
taskScheduler.start();
...
}
```

可以看到，JobTracker 此处通过 offerService() 函数调用 taskScheduler.start() 中的 start() 方法，就成功进入了我们实现 MesosScheduler 这个 Java 对象的 start() 方法中。那么在 start() 方法中都做了些什么呢？我们来看一下它的代码：

```
public void start() throws IOException {
    LOG.info("Starting MesosScheduler");
    jobTracker = (JobTracker) super.taskTrackerManager;

    Configuration conf = getConf();
    String master = conf.get("mapred.mesos.master", "local");

    this.eagerInitLitener = new erTaskInitializationListener(conf);
    eagerInitListener.setTaskTrackerManager(taskTrackerManager);
    eagerInitListener.start();
    taskTrackerManager.addJobInProgressListener(eagerInitListene

    frameworkScheduler = new FrameworkScheduler(this);
    driver = new MesosSchedulerDriver(
        frameworkScheduler, meworkScheduler.getFrameworkInfo(), master);

    driver.start();
}
```

首先从配置文件 mapred-site.xml 中读取到了 Mesos-master 的 IP 地址和端口号；然后为 JobTracker 添加了一个监听类，使提交的 Job 能够被即时地分解成 Task；最后 MesosScheduler 创建了一个 MesosSchedulerDriver 对象，并将自己作为参数传入该对象。该 MesosSchedulerDriver 对象是由 Mesos 本身的设计者用 C++ 实现的一个通用驱动器类，这个类的设计非常关键，它是整个 Mesos 架构体系的核心类，所有计算框架的调度器都是通过该驱动器连接进入 Mesos 的，它规定了所有调度器可以使用的接口函数、它的具体定义及各个接口的含义。

那么接下来 MesosSchedulerDrive.start() 做了哪些工作呢？首先以 MesosSchedulerDriver、FrameworkScheduler、Master 的 URL 等作为参数创建了一个 SchedulerProcess 对象，该对象其实是 LibProcess 的一个实例，初始化为一个收发消息的服务器，并且安装了各种消息处理函数。该对象用来与 Mesos-master 进行交互，响应 MesosSchedulerDriver 中调度资源、作业信息等请求。

与此同时，在 process::initialize() 中会以该 Process 的 PID 和 Mesos-master 的地址为参数创建一个 MasterDetector 对象。该对象用来监控 Master 状态（如新 Master 的选取、Master 的退出等）的变化，并且将这些变化的信息传递给该 PID，这样系统就可以随时掌握 Mesos-master 的状态。

MasterDetector 对象一旦被创建，就表明存在 Master 节点，它就会向 SchedulerProcess 发送一个 NewMasterDetected 消息。该对象会常驻内存，负责监控 Master 的状态，一旦状态有变便告知 SchedulerProcess 作相应的处理。

SchedulerProcess 收到 MasterDetector 发来的 NewMasterDetected Message 消息之后，会向 Mesos-master 发送一个 RegisterFramework Message 消息。Master 接收到该消息之后，会调用对应于该消息的处理函数——registerFramework() 方法，为该 Framework 按照一定的规则生成一个 Framework ID，创建一个 Framework 对象。

Master 接收到该 RegisterFramework Message 之后，为该框架产生一个 ID，并且将框架的信息（最主要的是框架的调度进程 SchedulerProcess 的 PID）放入自己的 hashmap<FrameworkID, Framework*>frameworks 之中，并且通知 Allocator 给这个框架分配资源。

5.4 Mesos 与 YARN 对比

首先，二者最大的不同点在于它们所采用的 Scheduler：Mesos 让 Framework 决定 Mesos 提供的这个资源是否适合该 Job，从而接受或者拒绝这个资源；而对于 YARN 来说，决定权在于 YARN，由 YARN 本身（自行替应用程序做主）决定这个资源是否适合该 Job。对于各种各样的应用程序来说，或许这就是个错误的决定。所以从扩展的角度来说，Mesos 更具有可扩展性。

其次，YARN 是 MapReduce 进化的产物，从诞生之日起就是为 Hadoop Jobs 管理资源的（YARN 也开始朝着 Mesos 涉及的领域进军），但它只为 Hadoop Jobs 提供了一个 Static Partitioning；而 Mesos 的设计目标是为各个框架（Hadoop、Spark、Web Services 等）提供 Dynamical Partitioning，让各个集群框架共用数据中心机器。

为了让大家更容易理解 Mesos 和 YARN 在架构上的相似性，下面给出 Mesos 和 YARN 的组件对应表，如表 5-1 所示。

表 5-1　Mesos 和 YARN 组件对应表

Mesos 中的组件	YARN 中的组件	功　能
Mesos-master	ResourceManager	整个集群的资源管理和调度
Mesos-slave Framework Executor Process	NodeManager	单个节点的资源管理（资源隔离、汇报等）、任务启动等
Framework Scheduler Process	ApplicationMaster	单个应用程序的管理和资源二次调度，基本操作均包括注册、资源申请 / 获取、资源分配（给内部的任务）等

既然 Mesos 和 YARN 如此相近，那么我们到底应该使用哪一个呢？或者说，哪一个系统更有前景呢？

就目前来看，YARN 在以下几个方面存在着明显的优势：（1）人力投入大，目前 YARN 有专门的公司（Hortonworks）负责维护和开发；（2）知名度高，YARN 是从 Hadoop 1.0 演化而来的，继承了 Hadoop 的知名度，且有大量公司和开发人员共享 Patch。然而，Mesos 最大的优点是设计简单、容易上手使用，不像 YARN 那样，一个资源的分配过程要涉及若干个状态机，且每个状态机有十几种状态、十几种事件。但从稳定性来看，两个系统都处于研发和测试阶段，离稳定可用还有一段距离。

5.5　小结

本章主要分析了 Hadoop 的调度机制和两种资源管理模型：YARN 和 Mesos。YARN 是一种新的 Hadoop 资源管理器，它是一个通用资源管理系统，可为上层应用提供统一的资源管理和调度，它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大好处。Mesos 是一个开源集群资源管理和调度系统，类似的系统有 Google 的 Borg 和 Omega、腾讯的 Torca、Hadoop 的 YARN 等。这类系统的目的是在一个集群上支持多种计算模型。Mesos 目前支持的计算机模型比较广泛，如 Hadooop、MPI、Torque、Spark 等。

6 Hadoop 集群管理之道

6.1 Hadoop 集群管理与维护

6.1.1 Hadoop 集群管理

在 Hadoop 的集群配置中，大部分使用的是默认设置。如果想要提高整个集群的性能，可以考虑通过修改配置的方法实现。配置项大部分位于 core_site.xml、hdfs_site.xml、mapred_site.xml 这三个文件里。Hadoop 常用的配置文件如下。

- hadoop-env.sh：记录脚本要用的环境变量。
- core-site.xml：Hadoop 核心配置项。
- hdfs-site.xml：Hadoop 守护进程配置项，例如 NameNode、SecondaryNameNode 和 DataNode 等。
- mapred-site.xml：MapReduce 守护进程配置项，包括 JobTracker 和 TaskTracker。

Hadoop 的主要控制脚本有如下几个。

- start-dfs.sh：在本地节点启动 NameNode，在 Slave 文件指定的每个节点启动 DataNode，在 Master 文件指定的每个节点启动 SecondaryNameNode。
- start-mapred.sh：在本地节点启动 JobTracker，在 Slave 文件指定的每个节点启动 TaskTracker。

- start-all.sh：依次调用 start-dfs.sh 和 start-mapred.sh。
- stop-dfs.sh/stop-mapred.sh/stop-all.sh：前面的 start 脚本对应的关闭脚本。
- hadoop-daemon.sh：上述脚本调用它来启动和终止 Hadoop 守护进程。如果用户需要从其他系统或自己编写脚本控制 Hadoop 守护进程，则可以调用它。
- hadoop-daemons.sh：用于在多台主机上启动同一 Hadoop 守护进程。

Hadoop 支持为所有的节点采用同一套配置文件，这样管理会变得非常简单，不过这对于某些集群来说并不适合。例如在扩展集群时，如果新机器的硬件和现有机器不同，则需要为新机器创建一套新的配置文件，以充分利用新硬件的资源。所以，笔者推荐在每个节点上保存一套配置文件，并由管理员完成这些配置文件的同步工作。Hadoop 提供了一个基本工具来进行同步配置，即 rsync。

在这种情况下，需要引入“机器类”的概念，为不同的机器类分别创建一套配置文件。不过 Hadoop 并没有提供执行这个操作的工具，需要借助外部工具，例如 chef、Puppet、cfengine、bcfg2 等。

对于小型集群（几十个节点），可以将 NameNode、SecondaryNameNode、JobTracker 放到单独一台机器上；但对于大型集群，最好分别放到不同的机器上。

- NameNode：在内存中保存整个命名空间的所有文件和块元数据，它的内存需求很大。
- SecondaryNameNode：保存一份最新的检查点，记录文件系统的元数据，有助于在数据丢失或系统崩溃时恢复 NameNode 的元数据；它在大多数时候空闲，但它创建检查时的内存需求和 NameNode 差不多。一旦文件系统包含大量文件，单台主机可能无法同时运行 NameNode 和 SecondaryNameNode。
- JobTracker：在一个运行大量 MapReduce 作业的高负载集群上，JobTracker 会使用大量内存和 CPU 资源，因此它最好运行在一个专用节点上。

HDFS 控制脚本需要在 NameNode 机器上运行；MapReduce 控制脚本需要在 JobTracker 机器上运行。

6.1.2 Hadoop 集群维护

1. Hadoop 的日志与监控

所有 Hadoop 守护进程都会产生日志文件，这些日志文件非常有助于查明系统中发生的事件。

在默认情况下，Hadoop 生成的系统日志文件存放在 $HADOOP_INSTALL/logs 目录下，也可以通过 hadoop-env.sh 文件中的 HADOOP_LOG_DIR 来进行修改。一般建议修改默认值，使之独立于 Hadoop 安装目录，这样便于以后的升级。通常可以把日志文件存放在 /var/log/hadoop 目录中。实现的方法就是在 hadoop-env.sh 文件中加入一行：export HADOOP_LOG_DIR=/var/log/hadoop。如果日志目录不存在，则会首先创建该目录；如果创建失败，请检查 Hadoop 用户是否有权创建该目录。

Hadoop 日志文件的名称格式为“hadoop- 用户名 - 守护进程名 - 本地主机名 .log”。其中“用户名”部分对应 hadoop-env.sh 文件中的 HADOOP_IDENT_STRING 变量，可以通过修改该变量来修改日志文件中的用户名信息。

在故障排查过程中，临时设置日志级别非常有益。有两种方法可以设置日志级别：网页和命令行。例如，要将某台机器的 JobTracker 设置为 DEBUG 级别，方法分别如下。

- 访问 http://jobtracker-host:50030/logLevel，将org.apache.hadoop.mapred.JobTracker 属性设置为 DEBUG 级别。
- 命令行：hadoop daemonlog -setlevel jobtracker-host:50030 org.apache.hadoop.mapred.JobTracker DEBUG。

2. Hadoop 的内存管理

Hadoop 为各个守护进程（NameNode、SecondaryNameNode、JobTracker、DataNode、TaskTracker）统一分配的内存在 hadoop-env.sh 文件中设置，参数为 HADOOP_HEAPSIZE，默认大小为 1000MB。

在大部分情况下，这个统一设置的值可能并不适合。例如对于 NameNode 节点，1000MB 的内存只能存储几百万个文件的数据块的引用。如果用户想单独设

置 NameNode 节点的内存，则可以通过 HADOOP_NAMENODE_OPTS 参数来设置。

同样，可以通过 HADOOP_SECONDARYNAMENODE_OPTS 参数来设置 SecondaryNameNode 的内存，使得它与 NameNode 保持一致。当然，还有 HADOOP_DATANODE_OPTS、HADOOP_BALANCER_OPTS、HADOOP_JOBTRACKER_OPTS 参数供用户使用。

此外，TaskTracker 启动独立的子 JVM 以运行 Map 和 Reduce 任务，分配给每个子 JVM 的内存量由 mapred.child.java.opts 属性（mapred-site.xml）控制，默认值为 200MB。

6.2 Hadoop 集群调优

6.2.1 Linux 文件系统调优

1. noatime 和 nodiratime 属性

文件挂载时设置这两个属性可以明显提高性能。在默认情况下，Linux Ext2/Ext3 文件系统在文件被访问、创建、修改时会记录下文件的时间戳，比如文件创建时间、最近一次修改时间和最近一次访问时间。如果系统运行时要访问大量文件，关闭这些操作，则可提升文件系统的性能。Linux 提供了 noatime 这个参数来禁止记录最近一次访问时间戳。

2. readahead buffer

调整 Linux 文件系统中预读缓冲区的大小，可以明显提高顺序读文件的性能。默认 buffer 大小为 256 sectors，可以增大为 1024 或者 2408 sectors（注意，并不是越大越好）。可使用 blockdev 命令进行调整。

3. 避免 RAID 和 LVM 操作

避免在 TaskTracker 和 DataNode 的机器上执行 RAID 和 LVM 操作，因为这通常会降低机器性能。所以在 Hadoop 集群中，一般禁用 Linux 逻辑管理卷（LVM）。

4. Linux 内核参数调整（/etc/sysctl.conf）

```
vm.swappiness = 0
```

6.2.2 Hadoop 通用参数调整

1. dfs.namenode.handler.count 或 mapred.job.tracker.handler.count

NameNode 或者 JobTracker 中用于处理 RPC 的线程数，默认是 10，较大集群可适当调大些，比如 64。

2. dfs.datanode.handler.count

DataNode 上用于处理 RPC 的线程数，默认为 3，较大集群可适当调大些，比如 8。需要注意的是，每添加一个线程，所需的内存就会增加。

3. tasktracker.http.threads

HTTP Server 上的线程数。运行在每个 TaskTracker 上，用于处理 Map Task 输出。大集群可以将其设为 40～50。

6.2.3 HDFS 相关配置

1. dfs.replication

文件副本数，通常设为 3，不推荐修改。

2. dfs.block.size

HDFS 中的数据块大小，默认为 64MB，较大集群可设为 128MB 或者 256MB（也可以通过参数 mapred.min.split.size 配置）。

3. mapred.local.dir 和 dfs.data.dir

这两个参数配置的值应当分布在各个磁盘目录上，这样可以充分利用节点的读 / 写能力。运行 Linux sysstat 包下的 iostat -dx 5 命令可以让每个磁盘都显示它的利用率。

4. dfs.DataNode.du.reserved

定义了每个 dfs.data.dir 所定义的硬盘空间需要保留的大小。以 Byte 为单位，默认值为 0，参考值为 10737418240。

5. dfs.NameNode.handler.count

定义 NameNode 工作线程池（用来处理客户端的远程过程调用及集群守护

进程调用）中的工作线程大小。一般将该值设为集群大小的自然对数乘以 20，即 20logN，N 为集群大小。

6. dfs.DataNode.failed.volumes.tolerated

定义整个 DataNode 声明失败前允许多少个硬盘出现故障。默认值为 0，参考值为 1。

7. fs.trash.interval(core-site.xml)

定义 Trash 目录下文件被永久删除前保留的时间。默认值为 0（不保存），推荐值为 1440（24 小时）。需要注意的是，垃圾回收功能只在命令行式有效，而对于 Java API 的删除则无效。

6.2.4 MapReduce 相关配置

1. MapReduce.tasks.maximum

同时运行在 TaskTracker 上的最大 MapReduce Task 数，一般设为 (core_per_node)/2~2×(cores_per_node)。

2. io.sort.factor

当一个 Map Task 执行完成后，本地磁盘上（mapred.local.dir）会有若干个 spill 文件，Map Task 最后做的一件事就是执行 merge sort，把这些 spill 文件合成一个文件（partition）。执行 merge sort 的时候，每次同时打开多少个 spill 文件由该参数决定。打开的文件越多，merge sort 不一定就越快，所以要根据实际情况适当调整。

3. mapred.child.java.opts

设置 JVM 堆的最大可用内存，需从应用程序角度进行配置。

4. mapred.java.child.opts

TaskTracker 是在一个独立的 JVM 进程里启动子任务的，启动时可以把一些参数传递给它。比较常见的是设置内存的使用。默认值为 200MB，参考值为 -Xms2g。

5. mapred.compress.map.output

启用 Map 端的压缩。参考值为 true。

6. mapred.map.output. compression.codec

定义了在 MapReduce 任务中使用的 Map 输出压缩码。如果该值为空，则 org.apache.hadoop.io.compress.defaultCodec 会被调用。参考值为 org.apache.hadoop.io.compress.SnappyCodec。

7. mapred.output. compression.type

如果 MapReduce 任务的输出需要写成 SequenceFile 格式，那么该参数会确定所使用的压缩类型。可配值有三种：第一种是记录（RECORD），该值会使得 SequenceFile 的每个数据都被单独压缩；第二种是块（BLOCK），该值会使得所有关键数据按固定大小分块压缩；第三种是无（NONE），设置该值就不会对数据进行压缩。参考值为 BLOCK。

8. mapred.job.tracker.handler.count

JobtRacker 会维护一个工作线程池来处理远程过程调用，该值用来控制线程池大小。参考值为 20logN，N 为集群大小。

9. Tasktracker.http.threads

控制每个 TaskTracker 用于处理请求的并行工作线程数。HTTP 线程的个数与集群 Reduce 槽的数量和被许可并行处理的数据份数是成正比的。

10. Mapred.reduce.slowstart.completed.maps

一旦一些 Map 任务生成了中间结果，就可以提前让 Reducer 对这些数据进行洗牌（Shuffle），尽可能早地复制数据，使 Reducer 任务在最后一个 Map 任务完成时直接运行。该属性定义允许启动 Reducer 进程时 Map 任务完成的百分比。参考值为 0.8，即 80%。

11. Mapred.reduce.parallel

Reduce Shuffle 阶段的 copier 线程数。默认值是 5，对于较大集群，可调整为 16～25。

12. mapred.reduce.parallel.copies

在 MapReduce 作业洗牌（Shuffle）过程中，每个 Reducer 任务需要从各个

TaskTracker 中获取同一作业运行 Map 任务产生的中间数据。该参数用于设置 Reducer 可以并行获取的数据个数。一般可以设置为 $4 \times \log N$，N 为集群大小。

6.2.5 Map 任务相关配置

1. io.sort.mb

该值是 Map 任务的输出结果和元数据在内存中所占的 Buffer 总大小。默认值为 100MB，对于大集群可设为 200MB。当 Buffer 达到一定阈值时，会启动一个后台线程来对 Buffer 的内容进行排序，然后写入本地磁盘（一个 spill 文件）。

Map 任务的一些输出会写在循环缓冲区内，该缓冲区大小是由该属性指定的。当该缓冲区的使用率达到 80% 时，就会启动一个后台进程将溢出的数据写到 mapred.local.dir 指定的路径下。

2. io.sort.spill.percent

这个值就是上述 Buffer 的阈值，默认是 0.8，即 80%。当 Buffer 中的数据达到这个阈值时，后台线程就会对 Buffer 中已有的数据进行排序，然后写入磁盘。

3. io.sort.record

它定义了一次性能合并的文件数目。以下两种情况会触发 MapReduce 的文件合并操作：第一种是 Map 任务执行完毕后 spill 文件需要合并；第二种是在 Reduce 获取所有 Map 任务输出文件后及用户调用 Reduce 代码之前。增大每一轮打开文件的个数进行合并可以减少数据读取和写入磁盘的时间，也减少了磁盘 I/O 操作。但合并更多的文件意味着需要更大的内存。

io.sort.mb 中分配给元数据的内存百分比默认是 0.05。这需要根据应用程序进行调整。

4. mapred.compress.map.output/ Mapred.output.compress

决定中间结果和最终结果是否要进行压缩，如果要进行压缩，则指定压缩方式（Mapred.compress.map.output.codec/ Mapred.output.compress.codec）。推荐使用 lzo 压缩。Intel 内部测试表明，相比未压缩，使用 lzo 压缩的 TeraSort 作业运行时间减少 60%，且明显快于 zlib 压缩。

6.2.6 HBase 搭建重要的 HDFS 参数

1. dfs.support.append

如果 HBase 需要持久存储到 HDFS，那么这个参数必须被明确设置为 true。

2. dfs.datanode.max.xcievers

默认值为 256，建议值为 2048。该值如果太小，则在客户端 I/O 负载密集时会报 I/O Exception。

6.3 Hadoop 集群监控

监控是系统管理的重要内容。监控的目的在于检测集群在何时未提供所期望的服务。主守护进程是最需要被监控的，包括主 NameNode、辅助 NameNode 和 JobTracker。DataNode 和 TaskTracker 经常出现故障，在大型集群中，故障率尤其高。因此，集群需要保留额外的容量，如此一来，即使有一小部分节点宕机，也不影响整个系统的运作。

管理员也可以定期运行一些测试作业，以检查集群的健康状况。

6.3.1 Apache Ambari 监控

Apache Ambari 是一种基于 Web 的工具，支持 Apache Hadoop 集群的供应、管理和监控。Ambari 目前已支持大多数 Hadoop 组件，包括 HDFS、MapReduce、Hive、Pig、HBase、ZooKeper、Sqoop、HCatalog 等。

Apache Ambari 支持 HDFS、MapReduce、Hive、Pig、HBase、ZooKeper、Sqoop、HCatalog 等的集中管理，也是 5 个顶级 Hadoop 管理工具之一。

Ambari 主要取得了以下成绩：

- 通过安装向导简化了集群供应。
- 预先配置好关键的运维指标（metrics），可以直接查看 Hadoop Core（HDFS 和 MapReduce）及相关项目（如 HBase、Hive 和 HCatalog）是否健康。
- 支持作业与任务执行的可视化及分析，能够更好地查看依赖和性能。
- 通过一个完整的 RESTful API 把监控信息暴露出来，集成了现有的运维工具。

- 用户界面非常直观，用户可以轻松、有效地查看信息并控制集群。
- Ambari 使用 Ganglia 收集度量指标，使用 Nagios 支持系统报警，当需要引起管理员的关注时（比如节点停机或磁盘剩余空间不足等问题），系统将向其发送邮件。

此外，Ambari 能够安装安全的（基于 Kerberos）Hadoop 集群，以此实现对 Hadoop 安全的支持。它同时提供了基于角色的用户认证、授权和审计功能，并为用户管理集成了 LDAP 和 Active Directory。

6.3.2 Ganglia 监控 Hadoop

当运行一个大型 MapReduce 作业时，我们通常非常关心该作业对 TaskTracker CPU、内存、磁盘及整个网络的带宽情况，这时候就需要类似 Ganglia 的工具为我们生成相关图表来诊断、分析问题。Ganglia 是一个监控服务器、集群的开源软件，能够用曲线图表现最近一小时、最近一天、最近一周、最近一月、最近一年的服务器或者集群的 CPU 负载、内存、网络、硬盘等指标。

Ganglia 的强大之处在于，Ganglia 集群中的任何其一台服务器端可收集到它下属的所有客户端数据，而且服务器端能够通过其中的任何一台客户端收集到同一网段的所有客户端数据。这个体系设计表示一台服务器能够通过不同的分层管理上万台机器。这个功能是其他如 MRTG、Nagios、Cacti 所不能比拟的。Ganglia 可以监控集群状态，但当服务器宕机或者某个 TaskTracker 出现故障时，它却无法通知用户，这时可以使用 Nagios 这款告警软件，它可以配置邮件告警和短信告警。通过编写插件，可以实现自己的监控功能。

可以把 Ganglia 的服务器端安装在 hadoop.online.master 上，把客户端安装在所有节点上。安装和配置 Ganglia 较为简单，不在此赘述。安装并配置完成后，启动 Ganglia 服务器端和客户端，以及 HTTP 服务器。

通过页面访问 Ganglia 的 GUI 界面：http://Ganglia_server_IP/ganglia。

6.4 小结

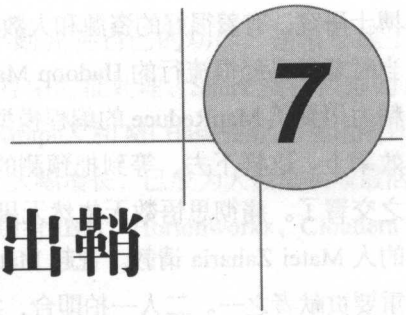
本章主要从经验层面阐述了 Hadoop 集群管理和维护的一些经验，给出了

Hadoop 集群调优的一些方法和参数，并介绍性地给出了 Hadoop 集群监控的两种方法。

Hadoop 集群管理和维护的问题很多，可能需要在实际应用中逐渐排除。Hadoop Log 对用户本身来说是一个很大的挑战，要解决用户遇到的问题，就需要去看日志，去分析 Hadoop 的源码，这些是最考验人的事。

第 2 篇

Spark 星火燎原



Spark 宝刀出鞘

7.1 Spark 的历史渊源

7.1.1 Spark 的诞生

任何一个强大事物的出现都有一个精彩有趣的历史背景，Spark 也不例外。Spark 最初的思想火花来自加利福尼亚大学伯克利分校（UC Berkeley）的一支研究团队。你可能会说，“哇，外国人在大数据处理方面着实很有创新力”。但是仅有创新力是不够的，必须要有创新的动机，才能造就今天 Spark 全球开花的局面。

说到 Spark 的创新动机，这里有个很有趣的故事。话说 Netflix（一家在线影片租赁公司）为了提高服务质量，更好地推荐适合用户口味的电影，搞了一个 Netflix 大奖赛，这个大奖赛从 2006 年 10 月份开始。Netflix 公开了 50 万用户针对 2 万部电影的大约 1 亿个匿名影片评级，数据集仅包含影片名称、评价星级和评级日期，没有任何文本评价的内容。大奖赛的目的在于预测 Netflix 的客户分别喜欢什么影片，要求把预测的准确度提高 10% 以上，以此来为用户推荐最适合的影片。最主要的是大奖赛的奖金为 100 万美元。

可谓重赏之下必有勇夫！一时间竟在 IT 界掀起一层不小的波浪，引来无数程序员跃跃欲试，如何处理这“1 亿个匿名影片评级”的大数据，摆在了众多参赛者

的面前。UC Berkeley 研究团队中有位名叫 Lester Mackey 的博士生，偶然的机会看到了这个悬赏，也打算试上一把。Lester 当时在 AMPLab 的大数据实验室进行博士研究，有着很好的资源和大数据处理能力。看到这个需求，Lester 马上想到了当时业界已经很流行的 Hadoop MapReduce。研究了一段时间，Lester 发现将很多精力用到了 MapReduce 的编程模型和低效的执行模式上，而不是花费在如何提高效率上。这样下去，等到把预测的准确度提高 10%，100 万美元奖金就与自己失之交臂了。痛彻思悟数天依然无果，便去找实验室的另一位专门研究分布式系统的人 Matei Zaharia 请教。说起 Matei，当时在业界已经小有名气，他是 Hadoop 的重要贡献者之一。二人一拍即合，共同分析了 Hadoop MapReduce 在此应用上的乏力之处，吸其精华，改其不足，定制化地输出了区区几百行代码，满足了 Lester 可以高效率地分布式建模的愿望。这就是 Spark 最初的版本。

看到这里，读者可能会想，拿着这么高大上的“宝剑”去参加 Netflix 武林大会，必赢无疑吧？但是我要遗憾地告诉你，Lester 所在的团队并没有抱得大奖归。虽然提高效率上和另一支团队的方案相差无几，并列第一，可是 Lester 所在的团队晚提交了 20 分钟，终和大奖擦肩而过。不过相比今天 Spark 给大数据处理方面带来的贡献，区区百万美元，实乃冰山一角。追溯此二人的发展，此后都成了学术界的杰出人物，Lester 成了斯坦福大学的教授，Matei 则成了麻省理工学院的教授，二人联合创办了 DatatBricks 公司，此公司正在主导并致力于 Spark 的高速发展。

7.1.2 Spark 的发展

2009 年，Spark 诞生于加利福尼亚大学伯克利分校的 AMPLab，最初属于伯克利分校的研究性项目。它于 2010 年正式开源，并于 2013 年捐赠给 Apache 来管理源码和后续发展，并于 2014 年成为 Apache 基金会的顶级项目。整个过程不到 5 年，发展速度如此之快，着实让人大为惊诧。如今已经逐渐形成了 Spark 生态圈，其对大数据的支持，从内存计算和流处理，到交互式查询，再到图计算和机器学习等，都在快速发展。如前文所述，在大数据处理方面，Hadoop MapReduce 已如日中天，是一种通用的大数据处理系统。还有诸如 Pregel、Storm、Griaph 等特殊化的大数据处理系统。Spark 作为后起之秀，不断地挑战 MapReduce，当然也会面临新产品的挑战。

关于 Spark 的发展历程，官方网站公布了每个 Release 版本新增的特性，这里

无须逐一展示。因为在很多时候人们更加关心的是 Spark 目前能做什么，以及不能做什么，而不是从什么时候可以做的。

Spark 以其 RDD 模型的强大表现能力，不断完善自己的功能，逐渐形成了一套自己的生态系统，其中主要包括 Spark 内存中的批处理、Shark 交互式查询、Spark Streaming 流式计算三大部分。此外还有 GraphX 和 MLBase 提供的常用图计算和机器学习算法。Spark 开源生态系统得到了大幅增长，已成为大数据领域最活跃的开源项目之一，当前主要贡献的公司和组织有 IBM、Hortonworks、Cloudera、MapR、Pivotal、华为、阿里巴巴、百度等。

7.2 Spark 和 Hadoop MapReduce 对比

一提到大数据处理，相信很多人第一时间想到的是 Hadoop MapReduce。没错，Hadoop MapReduce 为大数据处理技术奠定了基础。近年来，随着 Spark 的发展，越来越多的声音提到了 Spark。在业界有两种说法：一是 Spark 将代替 Hadoop MapReduce，成为未来大数据处理发展的方向；二是 Spark 将会和 Hadoop 结合，形成更大的生态圈。其实 Spark 和 Hadoop MapReduce 的重点应用场合有所不同。相对于 Hadoop MapReduce 来说，Spark 有点“青出于蓝”的感觉，Spark 是在 Hadoop MapReduce 模型上发展起来的，在它的身上我们能明显看到 MapReduce 的影子，所有的 Spark 并非从头创新，而是站在了巨人“MapReduce”的肩膀上。千秋功罪，留于日后评说，我们暂且搁下争议，来看看相比 Hadoop MapReduce，Spark 都有哪些优势。

1. 计算速度快

大数据处理首先追求的是速度。Spark 到底有多快？用官方的话说，“Spark 允许 Hadoop 集群中的应用程序在内存中以 100 倍的速度运行，即使在磁盘上运行也能快 10 倍”。可能有的读者看到这里会大为感叹，的确如此，在有迭代计算的领域，Spark 的计算速度远远超过 MapReduce，并且迭代次数越多，Spark 的优势越明显。这是因为 Spark 很好地利用了目前服务器内存越来越大这一优点，通过减少磁盘 I/O 来达到性能提升。它们将中间处理数据全部放到了内存中，仅在必要时才批量存入硬盘中。或许读者会问：如果应用程序特别大，内存能放下多少 GB？答曰：什么？GB？目前 IBM 服务器内存已经扩展至几 TB 了。

2. 应用灵活，上手容易

知道 AMPLab 的 Lester 为什么放弃 MapReduce 吗？因为他需要把很多精力放到 Map 和 Reduce 的编程模型上，极为不便。Spark 在简单的 Map 及 Reduce 操作之外，还支持 SQL 查询、流式查询及复杂查询，比如开箱即用的机器学习算法。同时，用户可以在同一个工作流中无缝地搭配这些能力，应用十分灵活。

Spark 核心部分的代码为 63 个 Scala 文件，非常的轻量级。并且允许 Java、Scala、Python 开发者在自己熟悉的语言环境下进行工作，通过建立在 Java、Scala、Python、SQL（应对交互式查询）的标准 API 以方便各行各业使用，同时还包括大量开箱即用的机器学习库。它自带 80 多个高等级操作符，允许在 Shell 中进行交互式查询。即使是新手，也能轻松上手应用。

3. 兼容竞争对手

Spark 可以独立运行，除了可以运行在当下的 YARN 集群管理外，还可以读取已有的任何 Hadoop 数据。它可以运行在任何 Hadoop 数据源上，比如 HBase、HDFS 等。有了这个特性，让那些想从 Hadoop 应用迁移到 Spark 上的用户方便了很多。Spark 有兼容竞争对手的胸襟，何愁大事不成？

4. 实时处理性能非凡

MapReduce 更加适合处理离线数据（当然，在 YARN 之后，Hadoop 也可以借助其他工具进行流式计算）。Spark 很好地支持实时的流计算，依赖 Spark Streaming 对数据进行实时处理。Spark Streaming 具备功能强大的 API，允许用户快速开发流应用程序。而且不像其他的流解决方案，比如 Storm，Spark Streaming 无须额外的代码和配置，就可以做大量的恢复和交付工作。

5. 社区贡献力量巨大

从 Spark 的版本演化来看，足以说明这个平台旺盛的生命力及社区的活跃度。尤其自 2013 年以来，Spark 一度进入高速发展期，代码库提交与社区活跃度都有显著增长。以活跃度论，Spark 在所有的 Apache 基金会开源项目中位列前三，相较于其他大数据平台或框架而言，Spark 的代码库最为活跃。

Spark 非常重视社区活动，组织也极为规范，会定期或不定期地举行与 Spark 相关的会议。会议分为两种：一种是 Spark Summit，影响力极大，可谓全球 Spark

顶尖技术人员的峰会，目前已于 2013—2015 年在 San Francisco 连续召开了三届 Summit 大会；另一种是 Spark 社区不定期地在全球各地召开的小型 Meetup 活动。Spark Meetup 也会在我国的一些大城市定期召开，比如北京、深圳、西安等地，读者可以关注当地的微信公众号进行参与。

7.3 Spark 的适用场景

从大数据处理需求来看，大数据的业务大概可以分为以下三类：

（1）复杂的批量数据处理，通常的时间跨度在数十分钟到数小时之间。

（2）基于历史数据的交互式查询，通常的时间跨度在数十秒到数分钟之间。

（3）基于实时数据流的数据处理，通常的时间跨度在数百毫秒到数秒之间。

目前已有很多相对成熟的开源和商业软件来处理以上三种情景：第一种业务，可以利用 MapReduce 来进行批量数据处理；第二种业务，可以用 Impala 来进行交互式查询；对于第三种流式数据处理，可以想到专业的流数据处理工具 Storm。但是这里有一个很重要的问题：对于大多数互联网公司来说，一般会同时遇到以上三种情景，如果采用不同的处理技术来面对这三种情景，那么这三种情景的输入 / 输出数据无法无缝共享，它们之间可能需要进行格式转换，并且每个开源软件都需要一支开发和维护团队，从而提高了成本。另外一个不便之处就是，在同一个集群中对各个系统协调资源分配比较困难。

那么，有没有一种软件可以同时处理以上三种情景呢？ Spark 就可以，或者说有这样的潜力。Spark 同时支持复杂的批处理、互操作和流计算，而且兼容支持 HDFS 和 Amazon S3 等分布式文件系统，可以部署在 YARN 和 Mesos 等流行的集群资源管理器上。

从 Spark 的设计理念（基于内存的迭代计算框架）出发，其最适合有迭代运算的或者需要多次操作特定数据集的应用场合。并且迭代次数越多，读取的数据量越大，Spark 的应用效果就越明显。因此，对于机器学习之类的“迭代式”应用，Spark 可谓拿手好戏，要比 Hadoop MapReduce 快数十倍。另外，Spark Streaming 因为内存存储中间数据的特性，处理速度非常快，也可以应用于需要实时处理大数据的场合。

当然，Spark 也有不适用的场合。对于那种异步细粒度更新状态的应用，例如 Web 服务的存储或增量的 Web 爬虫和索引，也就是对于那种增量修改的应用模型不适合。Spark 也不适合做超级大的数据量的处理，这里所说的“超级大”是相对于这个集群的内存容量而言的，因为 Spark 要将数据存储在内存中。一般来说，10TB 以上（单次分析）的数据就可以算是“超级大”的数据了。

一般来说，对于中小企业的数据中心而言，在单次计算的数据量不大的情况下，Spark 都是很好的选择。另外，Spark 也不适合应用于混合的云计算平台，因为混合的云计算平台的网络传输是很大的问题，即便有专属的宽带在云端 Cluster 和本地 Cluster 之间传输数据，相比内存读取速度来说，依然不抵。

7.4 Spark 的硬件配置

一般来讲，用户如果用 Spark Cluster 的形式来处理大数据，那么集群硬件配置自然较高。Spark 好比一匹宝马，我们要让马儿跑得快，就需要给马儿吃好料。

1. 内存的配置

虽然 Spark 是基于内存的迭代计算框架，但从官方资料来看，Spark 对内存的要求并不高，8GB 即可，当然这和 Spark Cluster 要处理的数据量的大小有关。一般情况下，内存容量越大越好，但这里要注意的是，如果内存太大，则可能需要特殊的配置，因为 JVM 对超大内存（256GB 以上）的管理存在问题。考虑到系统运行需要耗费内存，并且需要为 Application 的运行预留一些 Buffer（缓冲区），所以一般情况下为 Spark 应用分配 75% 的内存空间。如果需要处理超大规模的数据，则可以设置数据集的存储级别（后续讨论），以此来保证内存的高效、实用。

2. CPU 的配置

如果内存足够大，那么制约运算速度的可能就是 CPU 核数了。由于 Spark 实现的是线程之间的最小共享，可以支持一台机器扩展至数十个 CPU 核。对于目前服务器级别的机器来说，CPU 的配置一般在 16 核以上，完全可以满足 Spark 的运行要求。

3. 网络的配置

这里要特别说明的是，我们不建议将 Spark Cluster 部署到“混合型”的大数

据处理平台上，尽管这种“混合型”的大数据处理平台在本地很多云端都有专属的传输通道，但依然很难满足 Spark“低延迟”的需求。建议将 Spark Cluster 部署在 10GB 及以上网络带宽的局域网中，或者网络中拥有专用的大数据传输设备。

4. 存储的配置

从目前的硬件发展来看，存储硬盘的价格越来越低廉，性能越来越高效，借助一些大数据的存储方案，存储已不再是大数据处理的瓶颈所在。虽然 Spark 能够在内存中执行大量的计算，但它仍然需要本地硬盘作为部分数据的存储。Spark 官方推荐为每个节点配置 4 ～ 8 块磁盘，并且不需要配置为 RAID。另外，建议 Spark Cluster 存储本地化，可以通过配置 spark.local.dir 来指定磁盘列表。

7.5 Spark 架构

7.5.1 Spark 生态架构

如前所述，由于 Hadoop 缺少对迭代场景的支持和硬盘存储引起的延迟，让后起之秀 Spark 几乎在过去的一两年间“红透半边天”，Spark 和它的小伙伴们渐趋成熟稳定，配合也越来越默契，已经形成了 Spark Cluster 生态圈。好了，Spark 和他的小伙伴们在后台等待已久，现在通过图 7-1，让我们请 Spark 和它的小伙伴们集体亮相。

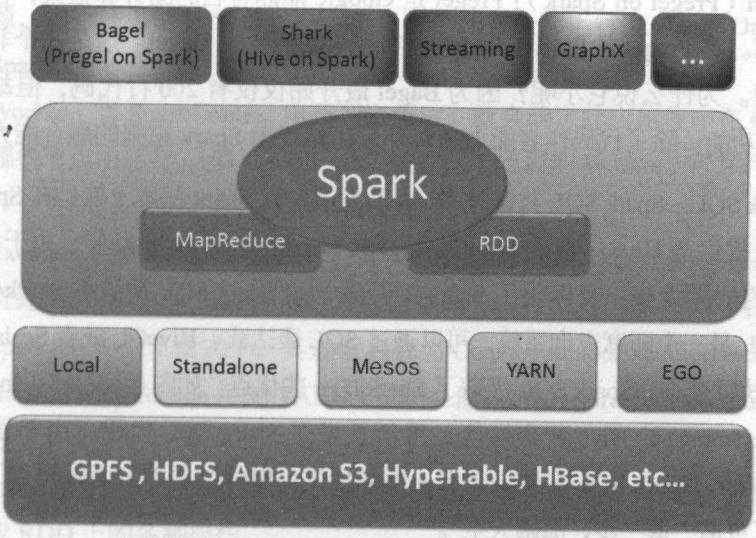


图 7-1 Spark 生态架构

读者可能会发现，有几个成员看着眼熟，比如 Hive、YARN、HDFS 等。是的，它们已经在 Hadoop 阵营中发展壮大，现在也为 Spark 的成长添砖加瓦，贡献自己的力量。我们先对 Spark 家族的主要成员进行简单介绍，后续章节我们再详细阐述。

从整个架构来看，最底层是存储类产品，比如之前详细介绍过的 HDFS、亚马逊的云存储服务 Amazon S3，Spark Cluster 可以通过各种 RESTful API 以编程方式实现与 S3 服务的交互。另外，耳熟能详的 Google 的 Bigtable 是一种高性能的、可伸缩的数据库技术，这里的 Hypertable 和 Bigtable 模型类似，具有高度可伸缩性，并且开源。可伸缩性是目前大数据处理平台越来越看重的技术之一。HBase 在之前详细介绍过，这里不再阐述。Spark 所用的存储类技术可谓遍地开花，不同的云平台提供不同的技术支持，后文我们还会专门讨论 IBM GPFS 技术。

有关 Spark 的运行模式多种多样，依托的资源分配技术也灵活多变。主要的运行模式有 Local、Standalone、Spark on Mesos、Spark on YARN、Spark on EGO 等。

RDD 可以说是 Spark 的“心脏”，也是最主要的创新点。其中文名称为“弹性分布式数据集”，它是一种只读的、分区记录的集合，每次对 RDD 数据集的操作结果都可以存放到内存中，下一个操作可以直接从内存中输入，省去了 MapReduce 大量的磁盘 I/O 操作。

Bagel（Pregel on Spark）：Pregel 是 Google 鼎鼎有名的图计算框架，在此不做过多介绍。将 Pregel 的思想在 Spark 上实现，便产生了 Bagel。这是一个非常有用的小项目。为什么说它小呢？因为 Bagel 最开始仅仅有 200 行代码，但却实现了 Pregel 的功能。除了代码作者能力超群外，还体现出 Spark 超强的融合能力。

Spark SQL：Spark SQL 其实是 Shark 的发展，而 Shark 基本上就是在 Spark 框架的基础上提供与 Hive 一样的 HiveQL 命令接口。随着 Spark 的发展，由于 Shark 对于 Hive 的依赖太强，便停止了 Shark 的开发，由 Spark SQL 来继承 Shark 的功能。Spark SQL 是一个即席查询系统，可以通过 SQL 表达式、HiveQL 或者 Scala DSL 在 Spark 上执行查询。Spark SQL 汲取了 Shark 的一些优点，如内存列存储（In-Memory Columnar Storage）、Hive 兼容性等，重新开发了 Spark SQL 代码。由于摆脱了对 Hive 的依赖，Spark SQL 在数据兼容、性能优化、组件扩展等方面都得到了极大的提升，真可谓“退一步，海阔天空”。

Spark Streaming：构建在 Spark 上处理流数据的框架，其基本原理是将 Stream 数据分成小的时间片段（秒级），以类似 batch 批量处理的方式来处理这一小部分数据。Spark Streaming 构建在 Spark 上，一方面是因为 Spark 的低延迟执行引擎（100ms+）可以用于实时计算；另一方面，相比基于 Record 的其他处理框架（如 Storm），RDD 数据集更容易做高效的容错处理。此外，小批量处理的方式使得它可以同时兼容批量和实时数据处理的逻辑与算法，方便了一些需要历史数据和实时数据联合分析的特定应用场合。

7.5.2 Spark 运行架构

1. Spark 运行的基本术语

我们首先需要搞清楚 Spark 中的几个概念和术语。可参看图 7-2 理解。

- Application：应用，其实就是用户需要完成的应用程序。一般来说，这部分代码需要用户根据自己的需求来完成。这部分代码主要包括两部分：Driver 功能部分和 Executor。那么问题来了：什么是 Driver？什么又是 Executor 呢？
- Driver：顾名思义，驱动者，为运行 Application 准备环境，驱动并监控 Application 运行。Application 中的 main() 函数在运行的时候便创建了一个 SparkContext（通常用 SparkConext 来代表 Driver）。SparkContext 初始化后，向资源管理器（可以是 Standalone、Mesos 或 YARN）注册并申请资源，进行任务的分配和监控等。当各个任务全部运行完成后，Driver 会将 SparkContext 关闭。
- Worker：当 SparkContext 申请到资源后，就会确定此应用程序在 Cluster 的哪些节点中运行。运行 Application 的这些节点就叫 Worker。在并行计算中，一个 Application 中会有多个 Worker 同时工作。
- Executor：顾名思义，执行者，其实就是运行在 Worker 上的一个进程，负责此 Worker 上的 Task 的运行及数据存储（内存或者磁盘）。举例来说，在 Spark on YARN 模式下，其进程名称为 CoarseGrainedExecutorBackend，和 Hadoop MapReduce 中的 YARN Child 类似。CoarseGrainedExecutorBackend 进程负责将 Task 包装成 TaskRunner，使其运行。
- Job：作业。多个并行计算的 Task 的集合，并且包含多个 RDD 及作用于相应 RDD 上的各种操作。

- Stage：阶段。每个 Job 会被拆分为很多组 Task，每组 Task 被称为 Stage，所以 TaskSet 是 Stage 的表现形式。
- Task：任务。它是在 Worker 上执行的最小单元。

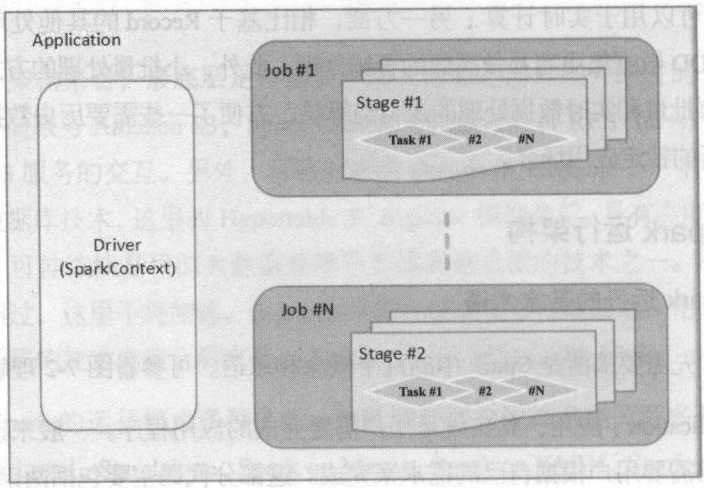


图 7-2　Spark 应用运行要素

2. Spark 运行流程

下面介绍一下 Spark Application 从开始运行到结束的整个过程，以便读者对 Spark 运行架构有一个整体的把握。将运行过程模型化，如图 7-3 所示（图中省略了诸多细节）。

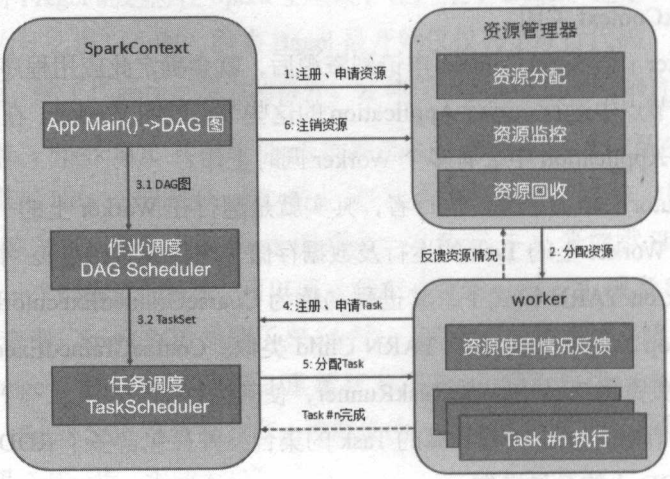


图 7-3　Spark 运行流程图

Spark具体的运行流程如下：

（1）构建Spark Application的运行环境。创建SparkContext后，SparkContext向资源管理器注册并申请资源。这里所说的资源管理器有Standalone、Messos、YARN等。事实上，Spark和资源管理器关系不大，主要是能够获取Executor进程，并能保持相互通信。在SparkContext初始化过程中，Spark分别创建作业调度模块DAGScheduler和任务调度模块TaskSchedule（此例为在Standalone模式下，在YARN-Client模式下任务调度模块为YarnClientClusterScheduler，在YARN-Cluster模式下任务调度模块为YarnClusterScheduler）。

（2）资源管理器根据预先设定的算法，在资源池里分配合适的Executor运行资源。在运行过程中，Executor运行情况将随着心跳发送到资源管理器上。考虑到Spark Application运行过程中SparkContext和Executor之间有大量的信息交换，提交SparkContext的Client应该靠近Worker节点，以方便信息传输。

（3）SparkContext构建DAG图（Directed Acyclic Graph，有向无环图），作业调度模块DAGScheduler将DAG图分解成Stage。DAGScheduler决定了运行Task的理想位置，并把这些信息连同Task本身传递给下层的TaskScheduler。

（4）Executor向SparkContext申请Task，告诉SparkContext，“我准备好了，可以干活了，请给我分配任务吧”。

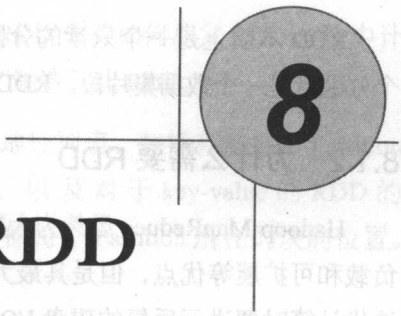
（5）TaskScheduler维护所有的TaskSet，当Driver收到Executor心跳的时候，TaskScheduler会根据其资源剩余情况分配相应的Task到Executor运行，同时SparkContext将应用程序代码发送给Worker，随后Task便开始在Worker上运行。在此期间，TaskScheduler还维护着所有Task的运行状态，重试失败的Task。如果Task失败是因为Shuffle数据丢失而引起的，则DAGScheduler需要重新提交运行之前的Stage；如果Shuffle数据没有丢失，则交由TaskScheduler处理。

（6）当Task运行结束后，反馈给SparkContext，并释放资源。

7.6 小结

本章回顾了Spark的诞生和发展过程。Spark犹如它的名字一样，自诞生以

来，便以星火燎原之势，迅速发展成大数据处理技术中的佼佼者。站在 Hadoop MapReduce 这个“巨人”的肩膀上，扬长避短，使用 RDD 来提升性能，满足交互式分析的需求，同时提升迭代算法的性能，这使得 Spark 非常适合迭代计算场景。总之，Spark 依然是大数据处理中的一个标杆。



8 Spark 核心 RDD

8.1 RDD 简介

8.1.1 什么是 RDD

之前我们已经多次提到 RDD，RDD 是 Spark 的核心成员，是 Spark 的思想精华所在。那么，RDD 到底是什么？它为何如此重要呢？

RDD 的全称为 Resilient Distributed Datasets，也有的资料缩写为 RDDs，它的中文名称是“弹性分布式数据集”。从它的名字中我们可以看到以下三点：

首先，RDD 是一个“数据集”。这个很容易理解。不过值得一提的是，RDD 是只读的、可分区的数据集。

其次，“分布式”的，也就是说，数据可以分布在 Cluster 中的多台机器上进行并行计算。有读者会问，这不就是“分布式共享内存”吗？其实它们还是有区别的，稍后我们会详细分析二者的不同。

最后，“弹性”的。有读者可能在想：“所谓弹性，可大可小，可伸可缩。我见过弹簧是弹性的，数据怎么也有弹性了呢？”在了解 RDD 之初，我们不妨这样理解这个“弹性”：在计算过程中，Spark 会根据内存的大小和使用情况来和磁盘进行数据交换，当内存不足时，它会将特定的部分数据写入磁盘；当数据块

被大量使用时，会将此数据块加载到内存。所以，相对于内存存储来说，数据是“弹性”的。

RDD 本质上是一个只读的分区记录集合，一个 RDD 可以包含多个分区，每个分区就是一个数据集片段。RDD 拥有很多操作，包括生成、转换和执行。

8.1.2 为什么需要 RDD

Hadoop MapReduce 虽然为大数据处理奠定了基础，并且具有自动容错、平衡负载和可扩展等优点，但是其最大的缺点是采用非循环式的数据流模型，使得在迭代计算时要进行反复的磁盘 I/O 操作，导致计算速度下降，使得 MapReduce 的性能大打折扣。

有的读者会想到，大量的磁盘 I/O 操作导致 MapReduce 性能下降，那把计算所需的数据块存储到内存中，问题岂不是解决了？对！这就是 Spark 解决问题的总体思路。

Spark RDD 可以 cache（缓存）到内存中，每次对 RDD 数据集的操作结果都可以存放到内存中，下一个操作可以直接从内存中输入，省去了 MapReduce 大量的磁盘 I/O 操作。这对于迭代运算比较常见的机器学习算法、交互式数据挖掘来说，效率提升比较大。这也是 Spark 存在的价值和发展迅速的理由。

8.1.3 RDD 本体的设计

读过《西游记》的人一定不会忘记唐僧的自我介绍：“贫僧自东土大唐而来，前往西天拜佛求经……”这大概是世界上最简明扼要的自我介绍了，他明确地说了“我是谁（贫僧）”、“从哪里来”、“到哪里去”、“要干什么”（下文我们改为“怎么来”）。我们让 Spark RDD 按照唐僧的思路来做个自我介绍。

从设计思路上看，每个 RDD 都要包含以下几部分。

（1）“我是谁”：数据集，包含一组数据分片（Partition，即数据集的基本组成单位）。

（2）“从哪里来”：关于 Lineage（血统）的信息，源码中的 Dependencies。我们可以简单地理解为它的“父母”是谁。这里需要说明的是，不一定每个 RDD 都有父 RDD。RDD 的生成有两种方式：一是父 RDD 通过 map 之类的函数转换而来；

二是从文件系统（比如 HDFS）输入创建，它是没有父 RDD 的，它的计算函数只是读取文件的每一行并作为一个元素返回给 RDD。

（3）“怎么来”：计算每个分片的函数（该 RDD 分片如何从父 RDD 分片中计算得到）。我们可以形象地理解为父 RDD 如何“制造”此 RDD。

（4）“到哪里去”：每个数据分片的预定义地址列表。包括存储每个 Partition 的优先位置（源码中的 preferredLocations），以及对于 key-value 的 RDD 的 Partitioner，或者对于一个 HDFS 文件来说，存储每个 Partition 所在的块的位置。此项为可选项。

8.1.4 RDD 与分布式共享内存

分布式共享内存（Distributed Shared Memory，DSM）是一种通用的内存数据抽象，使不同的机器无须共享物理内存就可以访问数据，进程访问 DSM 中的数据就如同访问本机的内存一样。

分布式共享内存和 RDD 相似的地方在于都将数据分布式地存储在内存中。二者的主要区别在于以下几点：

（1）RDD 可以批量操作（读操作或者转换创建），很多函数操作（如 count、collect 等）都是批量读操作，即扫描整个数据集，可以将任务分配到距离数据最近的节点上；而 DSM 只能通过细粒度的读操作和写操作。

（2）从容错机制来看，RDD 更容易实现而且快速（后面详细讨论）；而 DSM 则需要程序回滚（RollBack）。

（3）从一致性保护角度来看，RDD 无须此类保护机制，原因在于 RDD 是不可变的；而 DSM 需要保证在不同机器上执行的进程可以观察到他人对 DSM 的修改。

（4）RDD 还可以处理落后任务（即运行很慢的节点），这点与 MapReduce 类似；DSM 则难以实现备份任务，因为任务及其副本均需读 / 写同一内存位置的数据。

8.2 RDD 的存储级别

每个 RDD 可以使用不同的存储级别，可以把这个数据集存储在硬盘上，或者

以序列化的 Java 对象的形式存储在内存中，或者在多个节点上进行备份。RDD 根据“是否使用内存”、“是否使用硬盘”、“序列化”、“备份数”4 个参考项组合提供了 11 种存储级别（org.apche.spark.storage.StorageLevel），如表 8-1 所示。

表 8-1 RDD 存储级别列表

序　号	存储级别	参数	说明
1	NONE	StorageLevel(false, false, false)	
2	DISK_ONLY	StorageLevel(true, false, false)	以反序列化 Java 对象的形式存储在磁盘上
3	DISK_ONLY_2	StorageLevel(true, false, false, 2)	级别同上，但备份 2 份
4	MEMORY_ONLY	StorageLevel(false, true, true)	以反序列化 Java 对象的形式存储在内存，如果 RDD 太大不能完全放在内存，则多余的一些划分块不会被存储，会在需要时现生成
5	MEMORY_ONLY_2	StorageLevel(false, true, true, 2)	级别同上，但备份 2 份
6	MEMORY_ONLY_SER	StorageLevel(false, true, false)	以序列化的 Java 对象的形式存储在内存，这样比反序列化 Java 对象节省空间，但是耗费 CPU
7	MEMORY_ONLY_SER_2	StorageLevel(false, true, false, 2)	级别同上，但备份 2 份
8	MEMORY_AND_DISK	StorageLevel(true, true, true)	以反序列化 Java 对象的形式存储在内存，如果 RDD 太大不能完全放在内存，则把多余的划分块存储在硬盘上
9	MEMORY_AND_DISK_2	StorageLevel(true, true, true, 2)	级别同上，但备份 2 份
10	MEMORY_AND_DISK_SER	StorageLevel(true, true, false)	顾名思义，以序列化 Java 对象的形式存储在内存，多余的块存储在硬盘上
11	MEMORY_AND_DISK_SER_2	StorageLevel(true, true, false, 2)	级别同上，但备份 2 份

通过向 persist() 方法传递 org.apche.spark.storage.StorageLevel 来选择存储级别。调用的函数 persist() 如下：

```
private def persist(newLevel: StorageLevel, allowOverride: Boolean):
this.type = {
  if (storageLevel != StorageLevel.NONE && newLevel != storageLevel &&
!allowOverride) {
```

```
    throw new UnsupportedOperationException(
      "Cannot change storage level of an RDD after it was already assigned
a level")
    }
    if (storageLevel == StorageLevel.NONE) {
     sc.cleaner.foreach(_.registerRDDForCleanup(this))
     sc.persistRDD(this)
    }
    storageLevel = newLevel
    this
  }
```

默认的存储类别为 MEMORY_ONLY。

```
def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)
```

在存储类别为 MEMORY_ONLY 的情况下，CPU 的效率最高，从速度上来说也最有利于 RDD 的操作。一般来讲，内存使用率越大，CPU 效率越高。在选择存储类别时，应该兼顾二者。在非特殊情况下，非计算数据集的代价很高或者生成该数据集会过滤掉很多数据，否则不建议把数据存储在硬盘上，因为重新计算一个划分块比从硬盘中读取要快很多。当需要快速容错时，则使用带备份的存储级别。

RDD 的 cache() 方法类似于 MEMORY_ONLY，其实调用了 persist() 方法。

```
def cache(): this.type = persist()
```

8.3 RDD 依赖与容错

8.3.1 RDD 依赖关系

RDD 的容错能力很强，相对来说开销不大。在将 RDD 容错之前，我们需要明确 RDD 的依赖关系。通过依赖关系，我们可以看出 RDD 的容错能力是天生的，因为 RDD 容错很大程度上是从它和父 RDD 的关系中恢复的。

RDD 的依赖关系主要包括两种：窄依赖（Narrow Dependency）和宽依赖（Wide Dependency）。

窄依赖是指父 RDD 的每个分区最多被一个子 RDD 的分区所用。注意这里

的主体不是 RDD，而是 RDD 对应的 Partition（分区），而且这种依赖关系是不需要 Shuffle 的。在这类依赖中，可以根据 getParents() 方法获取某个 Partition 的父 Partitions。

```
@DeveloperApi
abstract class NarrowDependency[T](_rdd: RDD[T]) extends Dependency[T] {
 /**
  *为子分区获得相关的父分区
  * @参数 partitionId 代表子RDD的分区
  * 返回子RDD依赖的所有父RDD的分区
  */
 def getParents(partitionId: Int): Seq[Int]
 override def rdd: RDD[T] = _rdd
}
```

窄依赖有两种表现形式。

表现形式 1：表现为一个父 RDD 的分区对应一个子 RDD 的分区（OneToOneDependency），一个“上线”（父 RDD 的分区）对应一个“下线”（子 RDD 的分区）。例如，图 8-1 中的 map/filter 和 union 属于第一类。

```
@DeveloperApi
class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T](rdd) {
  override def getParents(partitionId: Int): List[Int] = List(partitionId)
}
```

表现形式 2：表现为两个父 RDD 的分区对应一个子 RDD 的分区（RangeDependency），两个“上线”对应一个“下线”。例如，图 8-1 中对输入进行协同划分（co-partitioned）的 Join 属于第二类。

```
@DeveloperApi
class RangeDependency[T](rdd: RDD[T], inStart: Int, outStart: Int, length: Int)
 extends NarrowDependency[T](rdd) {
 override def getParents(partitionId: Int): List[Int] = {
  if (partitionId >= outStart && partitionId < outStart + length) {
   List(partitionId - outStart + inStart)
  } else {
   Nil
  }
 }
}
```

窄依赖示例如图 8-1 所示。

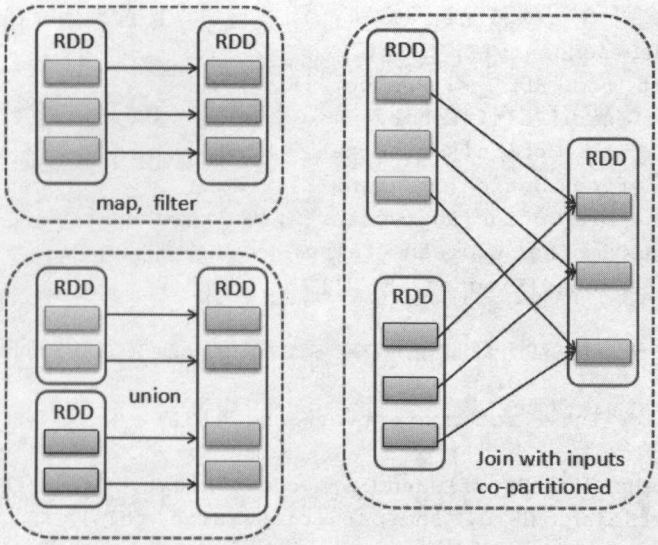


图 8-1　窄依赖示例

宽依赖是指子 RDD 的分区依赖父 RDD 的所有分区，这是因为 Shuffle 类操作，如图 8-2 中的 groupByKey 和未经协同划分的 Join。由于需要对父 Partition 进行划分，故需要用到 Shuffle，而 Shuffle 一般采用键值对的形式。宽依赖示例如图 8-2 所示。

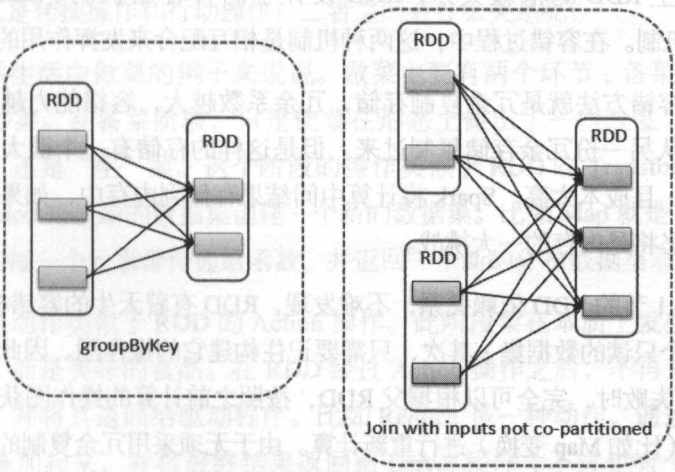


图 8-2　宽依赖示例

这里为每个 Shuffle 分配了一个全局唯一的变量 ShuffleId。为了进行 Shuffle，需要指定如何进行 Shuffle，这对应于参数 Partitioner；由于 Shuffle 是需要网络传输的，故需要进行序列化 Serializer。

```
@DeveloperApi
class ShuffleDependency[K, V, C](
  @transient _rdd: RDD[_ <: Product2[K, V]],
  val partitioner: Partitioner,
  val serializer: Option[Serializer] = None,
  val keyOrdering: Option[Ordering[K]] = None,
  val aggregator: Option[Aggregator[K, V, C]] = None,
  val mapSideCombine: Boolean = false)
 extends Dependency[Product2[K, V]] {

 override def rdd: RDD[Product2[K, V]] = _rdd.asInstanceOf[RDD[Product2[K, V]]]

 val shuffleId: Int = _rdd.context.newShuffleId()

 val shuffleHandle: ShuffleHandle = _rdd.context.env.shuffleManager.
registerShuffle(shuffleId, _rdd.partitions.size, this)

 _rdd.sparkContext.cleaner.foreach(_.registerShuffleForCleanup(this))
}
```

8.3.2 RDD 容错机制

基于以上 RDD 的依赖关系，RDD 设计了两种容错机制：Lineage 机制和 Checkpoint 机制。在容错过程中，这两种机制是相互配合来发挥作用的。

传统的容错方法就是冗余复制存储，冗余系数越大，容错能力越强。一块数据出错后，从另一份冗余存储复制过来。但是这样的存储有一个很大的问题：浪费存储空间，且成本太高。Spark 将计算中间结果存储到内存中，如果采用传统的容错方法，必将是内存的一大挑战。

回忆 8.3.1 节的 RDD 依赖关系，不难发现，RDD 有着天生的容错机制。首先，它自身是一个只读的数据集；其次，只需要记住构建它的操作图。因此，当执行任务的 Worker 失败时，完全可以根据父 RDD，按照之前计算的操作图获得之前执行的转换函数（比如 Map 变换）进行重新计算。由于无须采用冗余复制的方式支持容错，有效地降低了跨网络的数据传输成本，在容错速度和容错可靠度方面都有很大的优势。这种方法被称为“Lineage 机制”，只需要记录父 RDD 和它的操作图。

不过，在某些场景下，Spark 也需要利用 Checkpoint 的方式来支持容错。为什么呢？在窄依赖中，错误的子 RDD 可以直接通过计算父 RDD 的某个数据分

区来获得。但是宽依赖则要等到父 RDD 所有涉及的数据块都计算完成，并且父 RDD 的计算结果进行 Hash 计算并传递到对应节点上后才能计算子 RDD。比如 groupByKey 变换，子 RDD 中的数据块会依赖于多个父 RDD 中的数据分区，因为一个 key 可能错在父 RDD 的任何一个数据分区中。那么对于有宽依赖的并且长 Lineage 链的 RDD 来说，这种恢复非常耗时，适当地设置 Checkpoint 是很有必要的。另外，在 Spark Streaming 中，针对数据进行 update 操作，或者调用 Streaming 提供的 window 操作时，需要恢复执行过程的中间状态。此时，需要通过 Spark 提供的 Checkpoint 机制，以支持操作能够从 Checkpoint 中得到恢复。

如果看重容错的话，则可以使用带备份的存储类别（如 DISK_ONLY_2、MEMORY_ONLY_SER_2、MEMORY_AND_DISK_2）。虽然所有的存储级别都可以通过重新计算丢失的数据来达到容错，但是备份可以使得系统无须重新计算就能继续在 RDD 上执行 Task，效率显著提高。

8.4 RDD 操作与接口

RDD 操作主要有两大类：转换（Transformation）操作和行动（Action）操作。那么，什么是转换操作和行动操作？二者之间有什么关系呢？

我们举生活中做菜的例子来说说。做菜主要有两个环节：备菜（包括洗菜、切菜）和炒菜。在备菜阶段，只是使菜在形态上做出了一些改变，但本质是没有变化的，还是“生”菜。这个阶段的操作类似于 RDD 的 Transformation 操作。Transformation 是从新的数据集创建一个新的数据集。比如 Map 就是一种转换，它将数据集的每一个元素都传递给函数，并返回一个新的分布数据集表示结果。

而炒菜动作类似于 RDD 的 Action 操作。此阶段菜在本质上发生了变化，不再是生菜，而是美味的食品。在 RDD 经过 Action 操作之后，不再是数据集，而是一个值，并将其返回给驱动程序。比如 Reduce 是一种动作，通过一些函数将所有元素叠加起来，并将最终结果返回给 Driver 程序（不过也有个例，并行的 reduceByKey 能返回一个分布式数据集）。

需要特别说明的是，RDD 的 Transformation 操作是有“惰性”的，即不立即执行 Transformation 操作，需要 Action 操作来触发。好比我们不会无缘无故地去备菜，只有当炒菜这个动作来触发的时候，才会执行。RDD 的这种设计可以节省

很多无用的行为，可以更加高效地运行。驱动程序最终关注的是 Action 操作之后的结果（值），而不是中间数据集。例如，通过 map() 函数转换而来的新数据集（中间 RDD）将在 Count() 中使用。这个过程最终只返回 Count() 的结果给驱动程序，而不是整个过程数据集。

8.4.1 RDD Transformation 操作与接口

以下列举出源码中常用的 Transformation 操作，并对每个常用操作的功能进行简要说明。

- map 操作。

```
def map[U: ClassTag](f: T => U): RDD[U]
```

map 操作将 RDD[T] 中的每个元素进行函数运算、映射，生成一个新的 RDD[U]。

- filter 操作。

```
def filter(f: T => Boolean): RDD[T]
```

filter 操作将返回一个 RDD，包含了所有满足过滤条件的元素。

- distinct 操作。

```
def distinct(): RDD[T]
```

distinct 操作返回 RDD 中所有相异的元素，排除相同的元素。经过 distinct 操作之后，每个数据将只会出现一次。

- flatMap 操作。

```
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U]
```

flatMap 操作是将 RDD[T] 中的每个元素进行一对多转换，一个 T 可以映射成 0 到多个 U，两个 RDD 元素通常不等。

- sample 操作。

```
def sample(withReplacement: Boolean, fraction: Double, seed: Long =
Utils.random.nextLong): RDD[T]
```

sample 操作从数据中抽取一定比例的数据子集。

- union 操作。

```
def union(other: RDD[T]): RDD[T]
```

union 操作将两个 RDD[T] 合并成一个 RDD[T]。如果有相同的元素，并不会被排除，而是会多次显示（在实际操作中，可以使用 distinct() 函数将相同的元素排异）。

- coalesce 操作。

```
def coalesce(numPartitions: Int, shuffle: Boolean = false)(implicit
ord: Ordering[T] = null)
```

coalesce 操作对 RDD 的分区函数进行重新划分。

- repartition 操作。

```
def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]
```

repartition 操作是 coalesce 接口中 shuffle 参数为 true 时的实现。

- mapPartitions 操作。

```
def mapPartitions[U: ClassTag](f: Iterator[T] => Iterator[U],
preservesPartitioning: Boolean = false): RDD[U]
```

mapPartitions 操作是 Partition 级的转换，将每个 Partition 经过转换之后形成新的 RDD。

- groupByKey 操作。

```
def groupByKey(): RDD[(K, Iterable[V])]
```

groupByKey 操作将 RDD[(K,V)] 中所有的 (K,V) 键值对按 K 进行分组，每组一个元素，形成新的 RDD[(K, Iterable[V])]。

- reduceByKey 操作。

```
def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)]
```

在一个 (RDD[(K,V)] 中所有 (K,V) 键值对的数据集上使用，返回一个 (K,V) 键值对的 RDD，key 相同的值都被使用指定的 reduce() 函数聚合到一起。和 groupByKey 类似，任务的个数是可以通过可选参数来配置的。

- join 操作。

```
def join[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, W))]
```

将 RDD[(K,V)] 与 RDD[(K,W)] 执行 join 操作，形成新的 RDD[(K,(V,W))]，最终形成的 RDD 中只有两个 RDD 中共有的 K。

- mapValues 操作。

```
def mapValues[U](f: V => U): RDD[(K, U)]
```

仅对 RDD[(K,V)] 中的 V 进行转换，转换后和原先的 K 一起形成新的 (K,U) 键值对，通常和 groupByKey 一起使用。

- partitionBy 操作。

```
def partitionBy(partitioner: Partitioner): RDD[(K, V)]
```

partitionBy 操作将 RDD[(K,V)] 按照 K 进行重新分区。

8.4.2 RDD Action 操作与接口

- reduce 操作。

```
def reduce(f: (T, T) => T): T
```

reduce 操作通过函数 f 对 RDD[T] 中的元素进行两两合并，最终合并成一个值。这个函数必须是关联性的，确保可以被正确地并发执行。

- collect 操作。

```
def collect(): Array[T]
```

collect 操作以数组的形式返回数据集中的所有元素。

- count 操作。

```
def count(): Long
```

count 操作返回数据集的元素个数。

- take 操作。

```
def take(num: Int): Array[T]
```

take 操作返回一个包含数据集中前 *n* 个元素的数组。此操作目前并非在多个节点上并行执行，而是由 Driver 程序所在的机器单机计算所有的元素。

- first 操作。

```
def first(): T
```

first 操作返回数据集的第一个元素（类似于 take (1)）。

- saveAsTextFile 操作。

```
def saveAsTextFile(path: String): Unit
```

saveAsTextFile 操作将 RDD 的元素以 TextFile 的形式保存到分布式文件系统

（比如 HDFS 或者其他文件系统）。Spark 会调用 toString() 方法来把每个元素存成文本文件的一行。此方法最大的好处就是解决了通过 collect 到 Driver 再保存到磁盘的问题。

- saveAsSequenceFile 操作。

```
def saveAsSequenceFile[K, V, C <: CompressionCodec](
      pyRDD: JavaRDD[Array[Byte]],
      batchSerialized: Boolean,
      path: String,
      compressionCodecClass: String): Unit
```

将 RDD 的元素以 SequenceFile 的形式保存到分布式文件系统的指定目录（比如 HDFS 或者其他文件系统）。

- foreach(func) 操作。

```
def foreach(f: T => Unit): Unit
```

foreach(func) 操作对数据集中的每个元素逐个运行预定义的函数，比如用于更新一个累加器的变量。

8.5 RDD 编程示例

下面通过一个示例来说明 Transformation 与 Actions 操作在 Spark 中的使用。图 8-3 所示为将数据从文件系统中读入到处理完毕的 RDD 示例图（注：为了便于理解，省略了其他环节）。

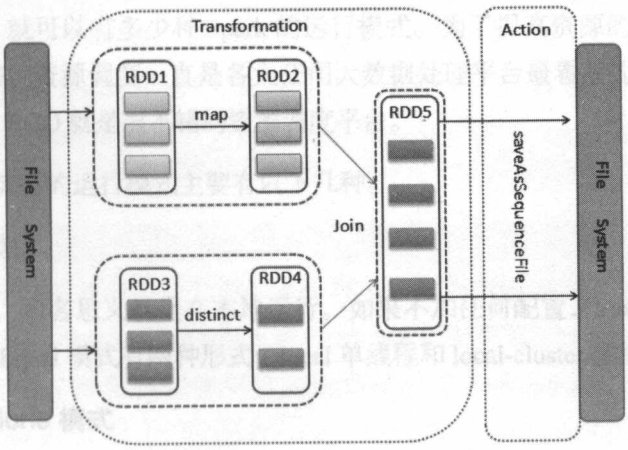


图 8-3 RDD 编程示例

```
val sourceData = new SparkContext(master, "Example", System.getenv("SPARK_
HOME"), Seq(System.getenv("SPARK_EXAMPLE_JAR")))

val rdd1 = sourceData.textFile(hdfs://path/to/data1)
val rdd2 = sourceData.textFile(hdfs://path/to/data2)

val rdd3 = rdd1.map(line => (line.substring(10), 1))
val rdd4 = rdd2.distinct()
val rdd5 = rdd4.join(rdd_5)

rdd5.saveAsSequenceFile(hdfs://save/as/path)
```

8.6 小结

RDD 作为 Spark 的核心，充分体现了 Spark 的主要设计思想，很大程度上解决了 Hadoop MapReduce 的不足之处。本章讲述了 RDD 的本体设计，阐述了 RDD 与传统的分布式共享内存技术的区别，介绍了 RDD 的永久性存储技术。本章重点讲述了 RDD 的依赖关系，以及从这种依赖关系中衍生出的容错机制。RDD 大体上有两种操作，即转换操作和行动操作，并列举了常用的操作和接口。总之，要想掌握 Spark 的精华，就必须先理解 RDD。



Spark 运行模式和流程

9.1 Spark 运行模式

9.1.1 Spark 的运行模式列表

Spark 越来越火，支持的运行模式也越来越多。那么为什么会有如此多的运行模式呢？主要是因为各个场景（或者各家公司）支持的资源调度模式不同。从总体上看，Spark 资源调度可以分为两大类：一类是自带的资源调度模块；另一类是外部的资源调度框架，比如 Mesos、YARN 等。所以从这个角度来说，有多少种资源调度框架，就可以有多少种 Spark 的运行模式。为了提高资源的利用率和大数据的处理效率，资源调度一直是各大公司大数据处理平台最看重的一块，比如 IBM Platform 的 EGO 就是很不错的资源调度平台。

Spark 目前流行的运行模式主要有以下几种。

1. Local 模式

Local 模式，顾名思义就是在本地运行。如果不加任何配置，Spark 默认设置为 Local 模式。Local 模式有两种形式：local 单线程和 local-cluster 多线程。

2. Standalone 模式

Standalone 模式也称为独立模式，在该模式中，Spark 集群由 Master 节点和

Worker 节点构成。用户程序通过与 Master 节点交互，申请所需的资源；Worker 节点负责具体 Executor 的启动运行。

3. Mesos 模式

Mesos 模式是采用 Apache Mesos 作为资源调度框架的一种模式，需要部署 Spark 和 Mesos 到相关节点。

4. YARN 模式

YARN 模式采用 YARN 作为资源调度框架。该模式主要分为两种：

- YARN-Cluster 模式，主程序逻辑和任务都运行在 YARN 集群中。
- YARN-Client 模式，主程序逻辑运行在本地，具体任务运行在 YARN 集群中。

5. EGO 模式

EGO 模式将 Spark 部署到 IBM Platform EGO 上，采用 Platform EGO 来整合硬件资源。

9.1.2 Local 模式

一般情况下，Local 模式不会在实际的大数据处理场景中使用，它主要用于代码的调试和快速验证，或者单机学习 Spark 等场景。运行 Spark 最简单的方法是通过 Local 模式。进入本地模式的方式如下：

```
./spark-shell --master local
./spark-shell --master local[2] # 本地运行，两个Worker线程，理想状态下为本地CPU 核数
```

在之前的 Spark 运行流程中，我们得知，SparkContext 是程序运行的总入口。在 SparkContext 的初始化过程中，我们可以通过 Master 来指定运行模式。SparkContext 的函数如下：

```
SparkContext sparkContext = new SparkContext(Master, appName, sparkHome, jars)
```

其中，Master 为运行模式，如果是本地模式，那么此值为 local，如果需要多个线程执行，则可以设置为 local[2]，表示两个线程；appName 为应用的名字；sparkHome 为 Spark 本地目录；jars 为第三方 JAR 包。

比如 SparkPi：

```
public class SparkPi {
     public static void main(String[] args) {
        SparkContext sparkConf = new SparkCont ("local" , "JavaSparkPi",
"/tmp/spark", "/tmp/spark-javaSparkPi.jar");
...
}
```

9.1.3 Standalone 模式

Standalone 模式也称为独立模式，其资源调度框架是由 Spark 自带的，是一种典型的 Master+Slave 集群结构。其中 Driver 运行在 Master 节点中，并且由常驻内存的 Master 进程守护；Worker 节点上常驻 Worker 守护进程，负责与 Master 通信，通过 ExecutorRunner 来控制运行在当前节点上的 CoarseGrainedExecutorBackend 进程。每个 Worker 上存在一个或者多个 CoarseGrainedExecutorBackend 进程。每个进程包含一个 Executor 对象，该对象持有一个线程池，每个线程可以执行一个 Task。

Spark Standalone Cluster 中计算模块的简要示意图如图 9-1 所示。

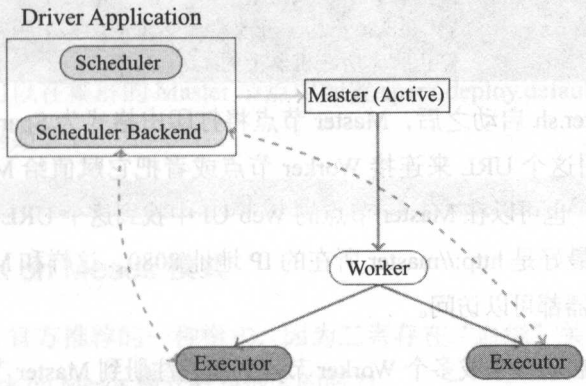


图 9-1 Spark Standalone Cluster 的计算模块示意图

从图 9-1 中可以看出，整个 Cluster 主要由 4 种不同的 JVM 组成：Master 负责管理整个 Cluster，Driver Application 和 Worker 都需要注册到 Master；Worker 负责某个节点上计算资源的管理，如启动相应的 Executor；Executor RDD 中每一个 Stage 的具体执行是在 Executor 上完成的；Driver Application 中的 SchedulerBackend 会因为部署模式的不同而不同。

换个角度来看，Master 对资源的管理是在进程级别，而 SchedulerBackend 对资源的管理则是在线程级别。启动时序如图 9-2 所示。

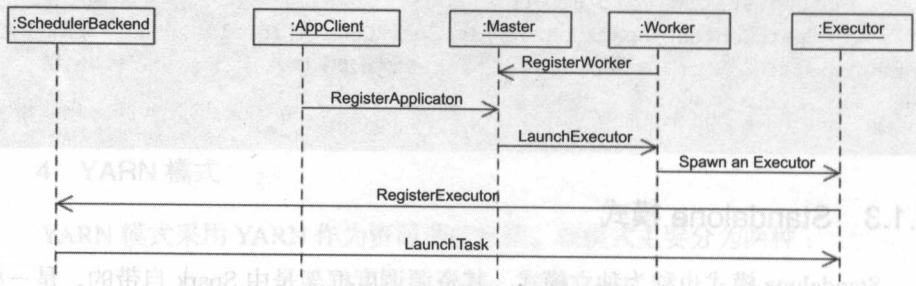


图 9-2　启动时序图

可以参照官方指导文档进行安装。安装好 Spark 后，在 sbin 目录下有启动脚本，可登录到 Master 上执行。

```
sbin/start-master.sh  启动脚本所在机器上的Master节点
sbin/start-slaves.sh  启动conf/slaves文件中指定的Slave所有节点
sbin/start-slave.sh   启动脚本所在机器上的Slave节点
sbin/start-all.sh     启动脚本所在的Slave节点及与其相关的Slave节点
sbin/stop-master.sh   停止脚本所在机器上的Master节点
sbin/stop-slaves.sh   启动conf/slaves文件中指定的Slave所有节点
sbin/stop-all.sh      停止脚本所在机器上的Master节点
```

在 start-master.sh 启动之后，Master 节点将打印出格式为 Spark://HOST:PORT 的 URL，可以用这个 URL 来连接 Worker 节点或者把它赋值给 Master 参数传递给 SparkContext。也可以在 Master 节点的 Web UI 中找到这个 URL，默认是 http://localhost:8080，最好是 http://master 所在的 IP 地址 8080，这样和 Master 节点在同一局域网内的机器都可以访问。

同样，可以启动一个或多个 Worker 节点并把它注册到 Master 节点上。执行如下命令：

```
./sbin/start-slave.sh <master-spark-URL>
```

一旦启动了 Worker 节点，通过 Master 节点的 Web UI，便可以看到其上注册的 Worker 节点的信息，比如 CPU 核数、内存等。

采用 spark-submit 脚本来提交应用。对于 Standalone 模式而言，Spark 目前支

持两种部署模式：在 Client 模式中，驱动程序在提交命令的机器上；在 Cluster 模式中，Driver 从集群的 Worker 节点中任取一个运行驱动程序。

在 Client 模式中的提交方法如下：

```
$SPARK_HOME/bin/spark-submit   --master spark://xa36502:7077 --class
org.apache.spark.examples.SparkPi     --deploy-mode          client
$SPARK_HOME/lib/spark-examples-1.4.1-hadoop2.6.0.jar 200
```

在 Cluster 模式中的提交方法如下：

```
$SPARK_HOME/bin/spark-submit   --master  spark://xa36502:6067 -   -class
org.apache.spark.examples.SparkPi     --deploy-mode           cluster
$SPARK_HOME/lib/spark-examples-1.4.1-hadoop2.6.0.jar 200
```

Standalone Cluster 模式的应用调度模式为 FIFO。为了运行多个用户，可以控制每个应用使用的最大资源。在默认情况下，它会使用集群中所有机器的核数，这只针对只有一个应用的集群有效。可以通过 spark.cores.max 参数来控制核数，代码如下：

```
val conf = new SparkConf()
         .setMaster(...)
         .setAppName(...)
         .set("spark.cores.max", "10")val sc = new SparkContext(conf)
```

另外，也可以在集群的 Master 节点中配置 spark.deploy.defaultCores 参数来改变默认值，代码如下：

```
export SPARK_MASTER_OPTS="-Dspark.deploy.defaultCores=<value>"
```

9.1.4 Spark on Mesos 模式

这是 Spark 官方推荐的一种模式，因为二者存在“血缘”关系，并且从发展趋势来看，Spark on Mesos 模式有着很大的潜力。

Spark on Mesos 模式有着很好的移植性，在 Spark 上开发算法程序，可以在本地模式调试成功后直接改用 Mesos 集群运行，除了文件的保存位置需要考虑外，算法理论上不需要进行任何修改。Spark 的本地模式支持多线程，有一定的单机并发处理能力，但不是很强劲。本地模式可以将结果保存在本地或者分布式文件系统，而 Mesos 模式必须保存在分布式或者共享文件系统。基于 Mesos 的 Spark 架构图如图 9-3 所示。

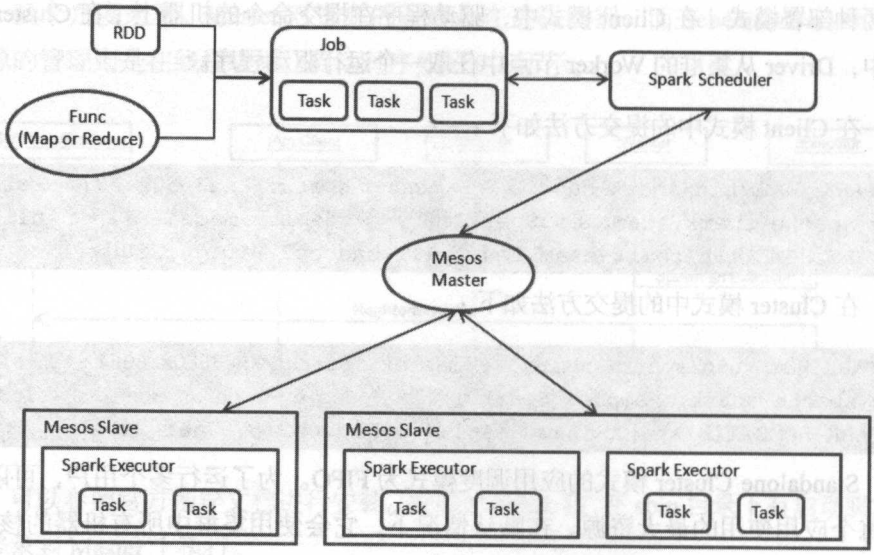


图 9-3　基于 Mesos 的 Spark 架构图

为了在 Mesos 框架上运行，需要安装 Mesos 的规范和设计。Spark 实现两个类：一个是 Spark Scheduler，在 Spark 中类名是 MesosScheduler；另一个是 Spark Executor，在 Spark 中类名是 Executor。有了这两个类，Spark 就可以通过 Mesos 进行分布式计算。Spark 会将 RDD 和 MapReduce 函数进行一次转换，变成标准的 Job 和一系列的 Task，并提交给 Spark Scheduler，Spark Scheduler 会把 Task 提交给 Mesos Master，由 Master 分配其不同的 Slave，最终由 Slave 中的 Spark Executor 将分配到的 Task 一一执行并且返回，组成新的 RDD，或者直接写入分布式文件系统。

目前在 Spark on Mesos 环境中支持两种模式：细粒度模式（Fine-grained Mode）和粗粒度模式（Coarse-grained Mode）。

1. 细粒度模式（Fine-grained Mode）

这是 Spark on Mesos 默认运行的模式，其典型思想是“按需分配”，用多少，分配多少。

在这种模式下，每个 Application 可以根据任务的运行情况在运行过程中动态获得更多或更少的资源，Mesos 会为每个 Executor 动态分配资源，每个 Executor 占用的资源仅仅是自己运行所需的资源，不需要考虑将来要运行的任务。所以这

种模式最大的优点是资源利用率高，并且容易控制。有利就有弊，此种模式的弊端在于，为了便于资源控制和更加细粒度的管理及容错，每个 Task 会将状态汇报给 Mesos Slave 和 Mesos Master，这会在每个 Task 启动的时候增加一些额外的开销。这种模式不适合于一些低延迟场景，例如交互式查询或者 Web 服务请求等。

2. 粗粒度模式（Coarse-grained Mode）

该模式的资源分配思想和细粒度模式截然不同。细粒度讲究动态分配，用多少，要多少。但是粗粒度模式会在应用程序的各个任务正式运行之前将其需要的资源申请好，然后开始运行任务。在运行过程中，这些资源一直持有（注意是持有，不是占有，或许资源处于空置状态），一直到整个 Application 结束后才会释放资源。

开启粗粒度模式的方法是在 conf/spark-default.conf 文件中进行如下配置：

```
spark.mesos.coarse true
```

在粗粒度模式下存在一个问题：如果一个 Spark Application 启动时会获取集群中所有的 CPU，这会导致在这个 Application 运行期间无法再运行其他任务。

可以通过控制一个 Application 获取到的最大资源来解决这个问题。解决办法是设置 CPU 的使用数，在 conf/spark-default.conf 文件中进行如下配置：

```
spark.cores.max 8
```

如此配置后，在提交任务后，Application 会一直占用 8 个 CPU，不会增加也不会减少，直到运行结束。

9.1.5 Spark on YARN 模式

Spark on YARN 模式分为 YARN-Cluster 和 YARN-Client 两种模式。

1. YARN-Cluster 模式

在 YARN-Cluster 模式中，Driver 在 Application Master 上运行，这意味着这个进程同时负责驱动应用程序和向 YARN 申请资源，并且这个进程在 YARN Container 中运行。启动应用程序的客户端程序可以在应用初始化后脱离。YARN-Cluster 模式如图 9-4 所示。

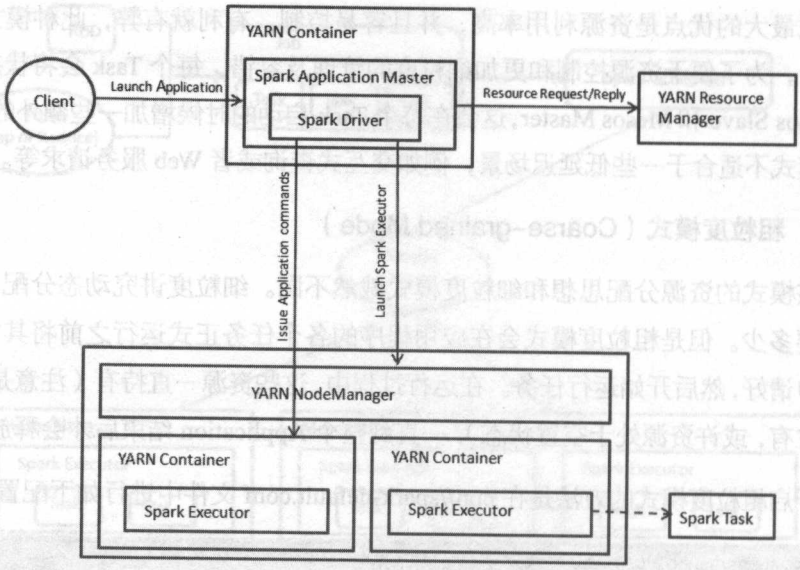


图 9-4　YARN-Cluster 模式

2. YARN-Client 模式

在 YARN-Client 模式中，Driver 运行在 Client 进程中，Client 一般不是集群中的机器。Application Master 仅仅用于向 YARN 申请 Executor Container。客户端进程在 Containers 启动之后与它们进行通信，调度任务。YARN-Client 模式如图 9-5 所示。

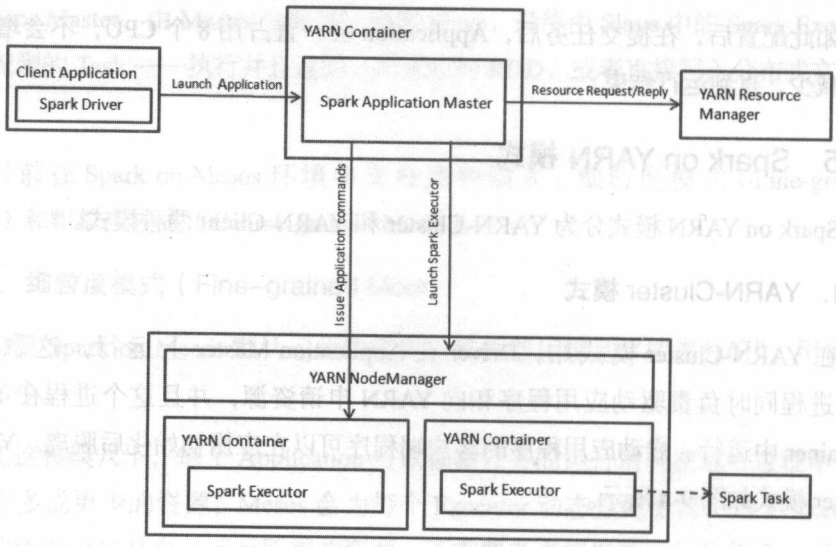


图 9-5　YARN-Client 模式

从结构图中我们可以明显地看出 YARN-Cluster 模式和 YARN-Client 模式的区别。首先是 Driver 运行的宿主不同，在 YARN-Cluster 模式中，Driver 运行在 Application Master 上；而在 YARN-Client 模式中，Driver 运行在 Client 上。其次，YARN-Cluster 模式不支持 Spark Shell，而 YARN-Client 模式支持。很明显，YARN-Cluster 模式用于产品环境较为合理；而 YARN-Client 模式更适合用在交互式调试场景中，它能让开发者实时看到应用程序的输出。

9.1.6 Spark on EGO 模式

在讨论 Spark on EGO 模式之前，我们先看看什么是 EGO。因为之前讨论的资源调度框架是开源的，比如 Mesos 或者 YARN，大家或多或少听说过。EGO 是 Enterprise Grid Orchestrator 的缩写，是 IBM Platform 的资源管理和调度、服务管理的工具。

在 Platform EGO 中，VEMKD 是 VEM 内核后台程序，运行在主控主机当中。它会启用其他后台程序并对分配请求做出响应。EGOSC 属于 EGO 服务控制器，负责向 VEMKD 申请相应资源并控制服务实例。流程处理管理器（简称 PEM）负责 VEMKD 中的启用、控制及监控活动，同时收集并发送运行时资源的使用情况。EGO 中的 Consumer 代表的是能够从集群处申请资源的一个实体。单一 Consumer 可以是业务服务、包含多种业务服务的复杂业务流程、单一用户或者一整条业务线。

和开源的 Spark 一样，Spark on EGO 使用相同的 DAGScheduler 和 TaskScheduler，架构图如图 9-6 所示。

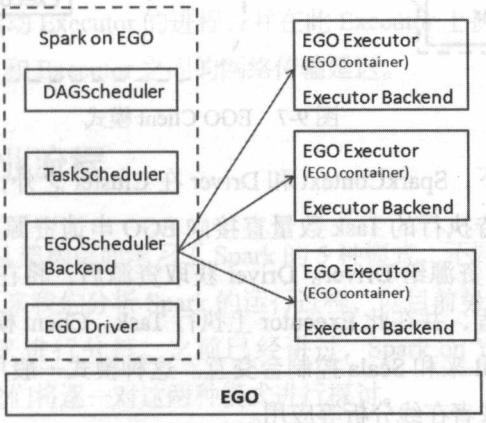


图 9-6　Spark on EGO 架构图

EGOSchedulerBackend 根据 TaskScheduler 提供的 Task 和 Task Stage 等信息，负责从目前的 EGO 框架中获取资源。用户可以自定义资源分配方案，并通过 Consumer 来分配资源。EGOSchedulerBackend 一旦获得资源，就可以通过 EGO Container 接口开始运行 Spark Executor。

EGOSchedulerBackend 监控 Spark Executor 运行的生命周期，以及资源使用情况和 Task 状态等。比如当 Task 完成时，EGOSchedulerBackend 触发调度逻辑来满足更多资源的获取或释放。

和 Spark on YARN 模式一样，Spark on EGO 也有两种模式：Client 模式和 Cluster 模式。

1. Client 模式

Client 模式的架构图如图 9-7 所示。

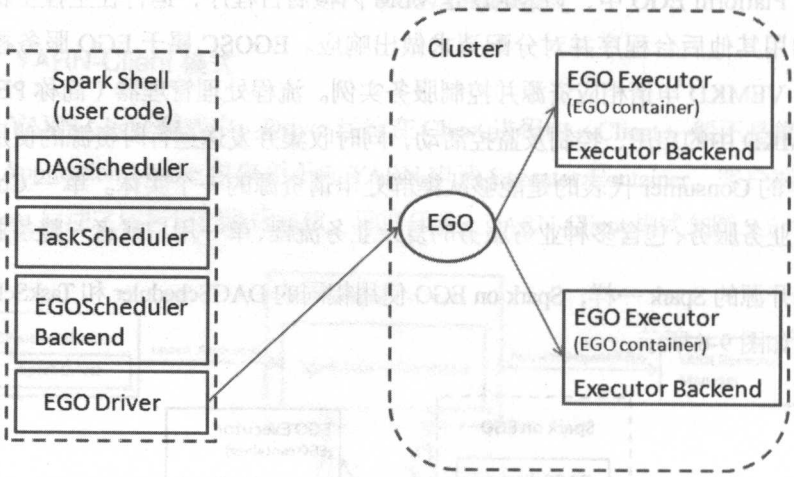


图 9-7　EGO Client 模式

在 Client 模式下，SparkContext 和 Driver 在 Cluster 之外，比如用户的笔记本电脑。Driver 根据待执行的 Task 数量直接向 EGO 申请资源，EGO 接到申请后，会分配 Cluster 中的资源给 Driver。Driver 获取资源后，将在 Cluster 中的节点上启动 Executor 的进程，并在此 Executor 上执行 Task。Client 模式支持 Spark Shell，可以通过 Spark Shell 来和 Scala 控制台交互。这种模式一般用来执行一些具有实时性要求的任务，或者在线分析等应用。

2. Cluster 模式

Cluster 模式的架构图如图 9-8 所示。

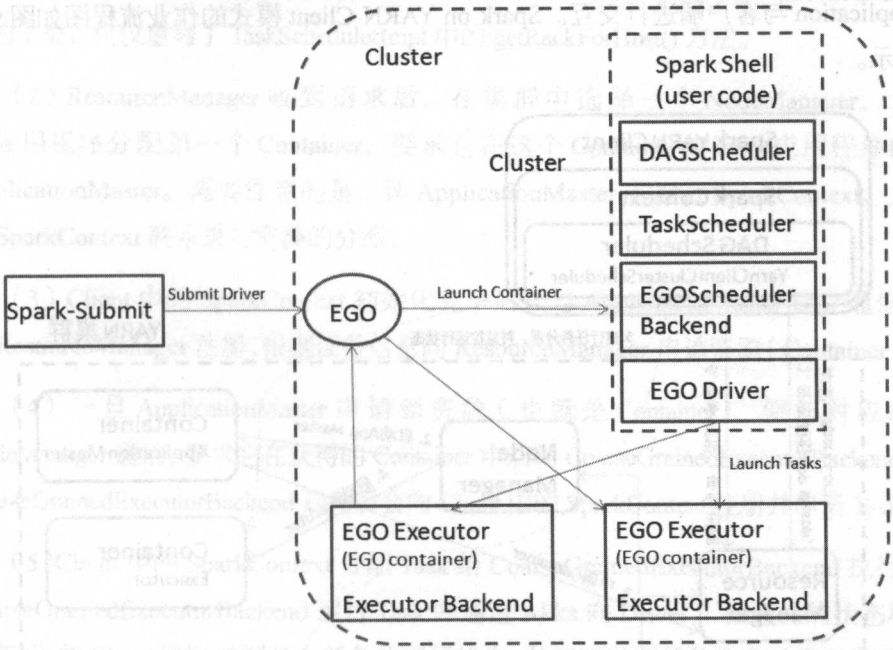


图 9-8 EGO Cluster 模式

在 Cluster 模式中，Spark Driver 在 Cluster 中，这和 Client 模式恰好相反。它通过 sbin/spark-submit 脚本提交 Application（一般为 .jar 或者 .py 文件），EGO 分配一个 Container 来启动 Driver。和 Client 模式类似，Driver 一旦启动，将在 Cluster 中的节点上启动 Executor 的进程，并在此 Executor 上执行 Task。使用这种模式可以减少 Driver 和 Executor 之间的网络传输延迟。

9.2 Spark 作业流程

9.1 节我们主要从架构层面学习了 Spark 的 5 种模式，不同的资源调度模块会有不同的模式。接下来我们分析 Spark 的运行过程，以目前势头最为强劲的 Spark on YARN 模式为例来进行分析。之前已经讲过，Spark on YARN 分为 Client 和 Cluster 两种模式，我们将逐一对这两种模式进行探讨。

9.2.1 YARN-Client 模式的作业流程

在 YARN-Client 模式中，Driver 在客户端本地运行，这种模式可以使得 Spark Application 与客户端进行交互。Spark on YARN Client 模式的作业流程图如图 9-9 所示。

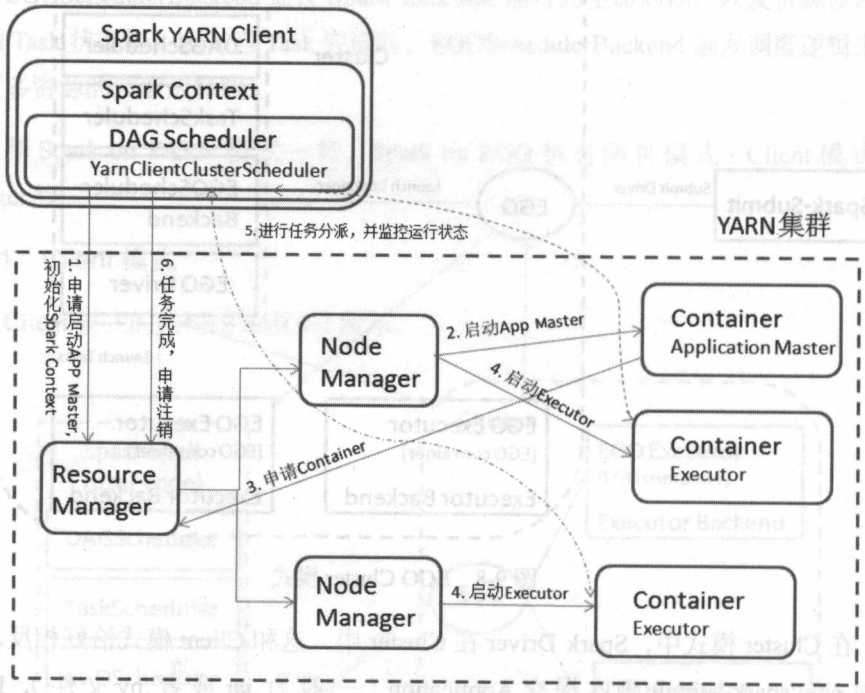


图 9-9　Spark on YARN Client 模式的作业流程图

其具体的作业流程如下：

（1）在 spark-submit 脚本提交 Application 之后，Spark YARN Client 向 YARN 的 ResourceManager 申请启动 Application Master。此提交过程将触发 SparkSubmit 类的 launch() 函数直接调用作业的 main() 函数，并对 SparkContent 进行初始化。在 SparkContent 初始化过程中，将进行相关的配置，注册 MapOutputTracker、BlockManagerMaster、BlockManager，创建 taskScheduler 和 DAGScheduler。其中比较重要的是创建 taskScheduler 和 DAGScheduler。在创建 taskScheduler 时会根据传进来的 Master 来选择 Scheduler 和 SchedulerBackend。由于是 YARN-Client 模式，程序会选择 YarnClientClusterScheduler 和 YarnClientSchedulerBackend，并

将 YarnClientSchedulerBackend 的实例初始化 YarnClientClusterScheduler，这两个实例的获取都是通过反射机制实现的，YarnClientSchedulerBackend 类是 CoarseGrainedSchedulerBackend 类的子类，YarnClientClusterScheduler 类是 TaskSchedulerImpl 类的子类，仅仅重写了 TaskSchedulerImpl 中的 getRackForHost() 方法。

（2）ResourceManager 收到请求后，在集群中选择一个 NodeManager，为该应用程序分配第一个 Container，要求它在这个 Container 中启动应用程序的 ApplicationMaster。需要注意的是，该 ApplicationMaster 不运行 SparkContext，只与 SparkContext 联系进行资源的分派。

（3）Client 中的 SparkContext 初始化完毕后，与 ApplicationMaster 建立通信，向 ResourceManager 注册，根据任务信息向 ResourceManager 申请资源（Container）。

（4）一旦 ApplicationMaster 申请到资源（也就是 Container），便与对应的 NodeManager 通信，要求它在获得的 Container 中启动 CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend 启动后会向 Client 中的 SparkContext 注册并申请 Task。

（5）Client 中的 SparkContext 分配 Task 给 CoarseGrainedExecutorBackend 执行，CoarseGrainedExecutorBackend 运行 Task 并通过 Akka 向 Driver 汇报运行的状态和进度，以便 Client 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动。在作业运行的时候，YarnClientSchedulerBackend 会每隔 1 秒通过 Client 获取到作业的运行状况，并打印出相应的运行信息，当 Application 的状态是 FINISHED、FAILED 或 KILLED 时，程序将退出等待。

（6）最后有个线程会再次确认 Application 的状态，当 Application 的状态是 FINISHED、FAILED 或 KILLED 时，程序运行完成。Client 的 SparkContext 向 ResourceManager 申请注销并关闭自己。

9.2.2 YARN-Cluster 模式的作业流程

在 YARN-Cluster 模式中，当用户向 YARN 中提交一个应用程序后，YARN 将分两个阶段运行该应用程序：第一阶段是把 Spark 的 Driver 作为一个 Application Master 在 YARN 集群中先启动；第二阶段是由 Application Master 创建应用程序，然后为它向 ResourceManager 申请资源，并启动 Executor 来运行 Task，同时监控它的整个运行过程，直到运行完成。值得一提的是，在此模式中，当作业提交到

YARN 上之后，客户端就空闲了，因为整个作业在 YARN 集群上进行，运行的结果将会保存到 HDFS 或者日志中。

Spark on YARN Cluster 模式的作业流程图如图 9-10 所示。

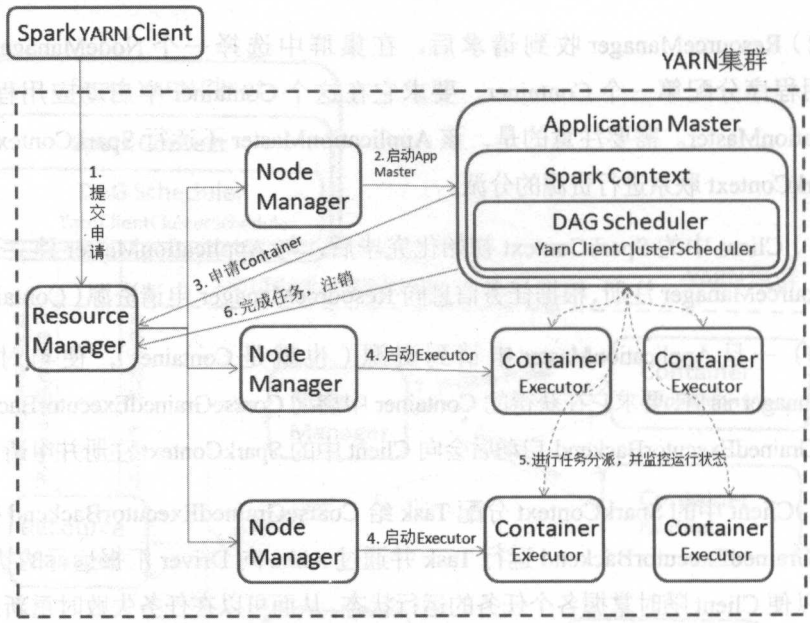


图 9-10　Spark on YARN Cluster 模式的作业流程图

其具体的作业流程如下：

（1）根据 YARN 配置信息来初始化，并启动 YARN Client，启动后向 YARN 中提交应用程序，包括 ApplicationMaster 程序、启动 ApplicationMaster 的命令、需要在 Executor 中运行的程序等。

（2）ResourceManager 收到请求后，在集群中选择一个 NodeManager，为该应用程序分配第一个 Container，要求它在这个 Container 中启动应用程序的 ApplicationMaster，由 ApplicationMaster 执行 SparkContext 等的初始化工作。在初始化 SparkContext 的时候将会创建 YarnClusterScheduler，在 SparkContext 初始化完成的时候会调用 YarnClusterScheduler 类中的 postStartHook() 方法，而该方法会通知 ApplicationMaster 初始化工作已完成。

（3）在 SparkContext、Driver 初始化完成后，ApplicationMaster 通过 amClient

向 ResourceManager 注册，这样用户可以直接通过 ResourceManage 查看应用程序的运行状态，然后采用轮询的方式通过 RPC 协议为各个任务申请资源，并监控它们的运行状态直到运行结束。

（4）一旦 ApplicationMaster 申请到资源（也就是 Container），便与对应的 NodeManager 通信，要求它在获得的 Container 中启动 CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend 启动后会向 ApplicationMaster 中的 SparkContext 注册并申请 Task。这一点和 Standalone 模式相同，只不过 SparkContext 在 Spark Application 中初始化时，使用 CoarseGrainedSchedulerBackend 配合 YarnClusterScheduler 进行任务调度，其中 YarnClusterScheduler 只是对 TaskSchedulerImpl 的一个简单包装，增加了对 Executor 的等待逻辑等。

（5）ApplicationMaster 中的 SparkContext 分配 Task 给 CoarseGrainedExecutorBackend 执行，CoarseGrainedExecutorBackend 运行 Task 并向 ApplicationMaster 汇报运行的状态和进度，以便 ApplicationMaster 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动。

在启动 Executor 之前，先要通过 yarnAllocator 获取到 numExecutors 个 Container，然后在 Container 中启动 Executor。如果在启动 Executor 的过程中失败的次数达到 maxNumExecutorFailures，则将 Application Status 标明为 PAILED，并关闭 SparkContext，此 Application 失败。其中 maxNumExecutorFailures 的计算规则如下：

```
// Default to numExecutors * 2, with minimum of 3
private val maxNumExecutorFailures = sparkConf.getInt("spark.
yarn.max.executor.failures",sparkConf.getInt("spark.yarn.max.
worker.failures", math.max(args.numExecutors * 2, 3)))
```

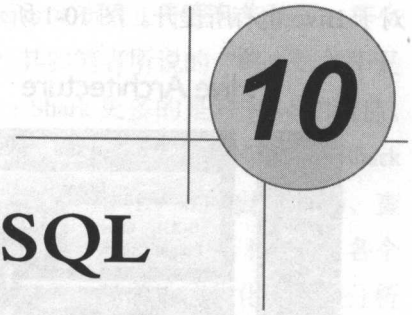
其实，启动 Executor 是通过 ExecutorRunnable 实现的，而 ExecutorRunnable 内部是启动 CoarseGrainedExecutorBackend 的。

（6）应用程序运行完成后，ApplicationMaster 向 ResourceManager 申请注销并关闭自己。

9.3 小结

本章分析了 Spark 目前用到的 5 种模式：Local 模式、Standalone 模式、Mesos

模式、YARN 模式和 EGO 模式。在较大的 Spark Cluster 中，用得比较多的是 Mesos 模式和 YARN 模式。总体而言，各种运行模式就是通过各种手段启动匹配的 SchedulerBackend 和 ExecutorBackend。除了 Local 模式和细粒度的 Mesos 模式外，其他模式最终都是通过基于 Akka 的 CoarseGrainedSchedulerBackend 和 CoarseGrainedExecutorBackend 完成任务调度的。



10 Shark 和 Spark SQL

10.1 从 Shark 到 Spark SQL

10.1.1 Shark 的撤退是进攻

Shark 这个名字在现在的 Spark 架构中已经看不到了，属于“昙花一现”。在 2014 年 7 月 1 日的 Spark Summit 上，Databricks 宣布终止对 Shark 的开发。但是不是说“一现”就没有了？Databricks 同时还宣布，Spark SQL 将涵盖 Shark 的所有特性，用户可以从 Shark 0.9 进行无缝升级，也就是说 Shark 的功能合并到 Spark SQL 中发展。所以为了更好地理解 Spark SQL，我们有必要对其前身 Shark 进行简单的了解。

Shark 是怎样诞生的呢？是为了解决什么问题呢？这要从 2011 年前说起。那时候，Hadoop MapReduce 处于如日中天的时代，其中有一个数据仓库工具 Hive（之前已经详细介绍过）的诞生，为 MapReduce 的易用性增添了不少色彩，它可以直接将 SQL 编译成可扩展的 MapReduce 作业。那么有人就想：能否在 Spark 中也引入 Hive 或者开发出类似 Hive 的工具呢？于是，Shark 就诞生了。

Shark 是由 UC Berkeley AMP 实验室的原班人马创建的 Databricks（商业化）开发，可以简单地理解为 Hive on Spark，本质上是通过 Hive 的 HQL 解析，把

HQL 翻译成 Spark 上的 RDD 操作，然后通过 Hive 的 MetaData 获取数据库里的表信息，HDFS 上的数据和文件会由 Shark 获取并放到 Spark 上运算。但是 Shark 相对于 Hive 也有所提升。图 10-1 所示是它们之间的架构对比。

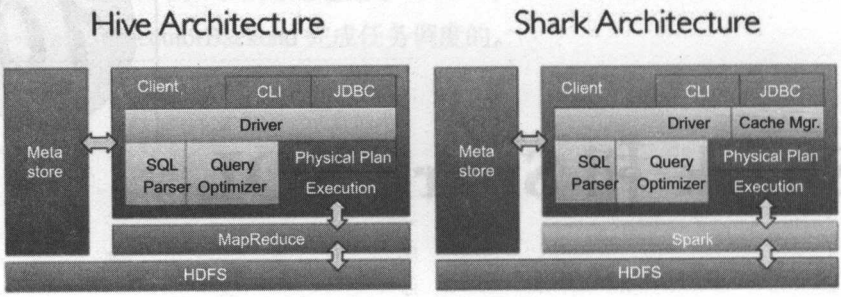


图 10-1　Hive 与 Shark 对比

在上面的架构图中，Shark 分为内存管理、物理计划、执行三个模块，并使之能够运行在 Spark 引擎上，从而使得 SQL 查询的速度得到 10～100 倍的提升。为了最大限度地保持和 Hive 的兼容性，Shark 使用 Hive 的 API 来实现 Query Parsing 和 Logic Plan Generation，最后的 Physical Plan Execution 阶段用 Spark 代替 Hadoop MapReduce。由于与 Hive 兼容，Shark 可以在任何支持 Hadoop 存储 API 的系统上进行查询，包括 HDFS 和 Amazon S3 等。Shark 也支持多种数据格式，例如 TEXT、JSON、XML 等。Shark 继承了 Hive 的 schema-on-read 和嵌套数据类型，并且可以在 Shell 模式下使用 rdd2sql() 这样的 API，把 HQL 得到的结果集继续在 Scala 环境下运算，支持自己编写简单的机器学习或简单分析处理函数，对 HQL 的结果进一步分析计算。通过配置 Shark 参数，Shark 可以自动在内存中缓存特定的 RDD，实现数据重用，进而加快特定数据集的检索。同时，Shark 通过 UDF 用户自定义函数实现特定的数据分析学习算法，使得 SQL 数据查询和运算分析能结合在一起，最大化 RDD 的重复使用。

另外，Shark 支持机器学习。Shark 选择 Spark 作为执行引擎、RDD 作为 Operators 操作的主要数据结构，故而 Shark 遵循将计算能力移到数据存储位置的原则来支持机器学习。Shark 除了执行 SQL 查询并返回其结果外，还允许查询返回代表了查询计划的 RDD，用户可以使用返回的 RDD 进行分布式计算。

Shark 会在集群上自动并行执行 map、mapRows 和 reduce 函数，并且 Master 程序将收集 reduce 函数的输出进行更新操作。Shark 提供了一些基本的机器学习算

法，包括 linear regression、logistic regression 和 k-means。在大部分情况下，用户只需要实现 mapRows 函数提取待分析的数据，然后调用 Shark 的基本学习算法。

刚才我们提到，Shark 已经被主导方 Databricks 宣布终止开发与支持了，那么就有人质疑了，为什么要终止这么好的东西呢？其实笔者所说的“终止”并不是舍弃，而是一种变革。这种“终止”的原因是：Shark 更多的是对 Hive 的改造，替换了 Hive 的物理执行引擎，因此运行速度更快。然而，不容忽视的是，Shark 继承了大量的 Hive 代码，对 Hive 的依赖性太强（如采用 Hive 的语法解析器、查询优化器等），制约了 Spark 的 One Stack Rule Them All 的既定方针和 Spark 各个组件的相互集成，给优化和维护工作带来大量的麻烦。随着性能优化和先进分析整合的进一步加深，基于 MapReduce 设计的部分无疑成了整个项目的瓶颈。因此，为了得到更好的发展，给用户提供更好的体验，Databricks 宣布终止 Shark 项目，从而将更多的精力转移到 Spark SQL 上，并在 Spark SQL 上支持 Shark 的功能。停止对 Shark 的支持，是为了更好地发展 Spark。

10.1.2 Spark SQL 接力

在 Spark 1.0.0 版本中，Spark SQL 从 Shark 手中接过沉重的“接力棒”，汲取了 Shark 的一些精华，如内存列存储（In-Memory Columnar Storage）、Hive 兼容性等，重新组织架构，并开发了 Spark SQL 代码。官方对 Spark SQL 的表述为：Spark SQL 是支持在 Spark 中使用 SQL、HiveQL、Scaca 的关系型查询表达式。它的核心组件是一个新增的 RDD 类型 SchemaRDD，它把行对象用一个 Schema 来描述行里面的所有列的数据类型，它就像关系型数据库里的一张表。它可以从原有的 RDD 创建，也可以是 Parquet 文件，最重要的是它可以支持用 HiveQL 从 Hive 里面读取数据。其实简单地理解，Spark SQL 是一个即席查询系统，可以通过 SQL 表达式、HiveQL 或者 Scala DSL 在 Spark 上执行查询；Spark SQL 引入了新的 RDD 类型 SchemaRDD，可以像传统数据库定义表一样来定义 SchemaRDD（由定义了列数据类型的行对象构成）。

值得一提的是，随着 Shark 画上句号，Spark 阵营中发展出了两个项目，其中一个是 Spark SQL，而另一个是 Hive on Spark。其中 Spark SQL 作为 Spark 生态圈的一员继续发展，而不再受限于 Hive，只是兼容 Hive；而 Hive on Spark 是 Hive 的一个发展计划，该计划将 Spark 作为 Hive 的底层引擎之一，也就是说，Hive 将

不再受限于一个引擎，可以采用 Map-Reduce、Tez、Spark 等引擎。Spark SQL 由于摆脱了对 Hive 的依赖，因而在以下方面得到了极大的提升。

（1）数据兼容方面：不但兼容 Hive，还可以从 RDD、Parquet 文件、JSON 文件中获数据，未来版本甚至支持获取 RDBMS 数据及 Cassandra 等 NoSQL 数据。

（2）性能优化方面：除了采用 In-Memory Columnar Storage、byte-code Genetion 等优化技术外，将引进 Cost Model 对查询进行动态评估、获取最佳物计划等。

（3）组件扩展方面：无论是 SQL 的语法解析器、分析器还是优化器，都可以重新定义并进行扩展。

实际上，笔者认为，“Spark SQL”这个名字并不恰当。虽然名字叫“Spark SQL”，但其应用并不局限于 SQL。根据 Spark 的官方表述，Spark SQL 是一个用于处理结构化数据的 Spark 组件。该定义强调的是“结构化数据”，而非“SQL”。Spark 1.3.0 更加完整地表达了 Spark SQL 的愿景：让开发者用更精简的代码处理尽量少的数据，同时让 Spark SQL 自动优化执行过程，以达到降低开发成本、提升数据分析执行效率的目的。为此，我们在 Spark 1.3.0 中引入了与 R 和 Python Pandas 接口类似的 DataFrame API，延续了传统单机数据分析的开发体验，并将之推广到了分布式大数据场景。

10.1.3 Spark SQL 与普通 SQL 的区别

Spark SQL 与普通 SQL 的主要区别有：

（1）由于 Spark SQL 查询基于 Spark 平台，所以速度会比普通 SQL 有极大的提升，并且与 Hive 相比性能也有较大的提升。因为 Hive 原生是基于 MapReduce 的，生成 MapReduce Job，而 Hive on Spark 生成 Spark Job，充分利用 Spark 的快速执行能力来缩短 HiveQL 的响应时间。

（2）内存在列存储 SQLContext 下 cache/uncache table 的时候会调用“列存储模块”。该模块借鉴自 Shark，目的是当把表数据缓存在内存中的时候执行行转列操作，以便压缩。Spark SQL 创建的表会变成内存缓存表，对象变少，GC 变少，下次需要表中数据的时候可以使用上次缓存的表，因此效率变快。普通 SQL 在执行过程中有时不需要读取物理表就可以返回结果，比如重新运行刚运行过的 SQL 语句，可能直接从数据库的缓冲池中获取返回结果。

（3）Runtime Bytecode Generation，根据查询动态生成代码。当我们进行查询的时候，普通 SQL 被处理的时候会生成复杂的树；而 Spark SQL 通过自己的解释引擎，结合 Scala 语言的 Runtime Reflection 特性极大地简化了 SQL 语句的处理。

（4）Spark SQL 的查询会变成 Schema RDD，这样就可以执行 RDD 的很多操作，比如 count。这样的特性带来了无限的可能性。而普通的 SQL 就不可能有这样的性质。

（5）能在 Scala 代码里面写 SQL，支持简单的 SQL 语法检查，能把 RDD 指定为 Table 存储起来，以此完成实时的数据交互。此外，支持部分 SQL 语法的 DSL。

（6）Spark SQL 支持 Parquet 文件的读 / 写，并且保留 Schema，支持列存储。除此之外，可以更好地使用 JDBC 和 ODBC，因此非常有利于 BI Tools。支持 jsonFile 和 jsonRDD，读取 JSON 文件之后，转换成 SchemaRDD。这些支持丰富了 Spark SQL 的数据接入场景。

10.2 Spark SQL 应用架构

Spark SQL 的应用架构如图 10-2 所示。

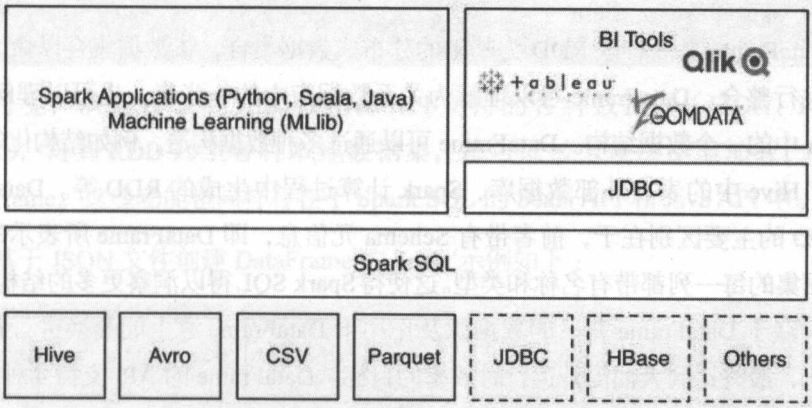


图 10-2 Spark SQL 的应用架构

从图 10-2 中可以看出 Spark SQL 的重要性。DataFrame 是 Spark SQL 的核心，它将数据保存为行构成的集合，行对应列有相应的列名。使用 DataFrame 可以非常方便地查询数据、给数据绘图及进行数据过滤。DataFrame 也可以用于数

据的输入与输出，例如利用 Spark SQL 中的 DataFrame 可以很容易地将很多数据格式加载为表并进行相应的查询操作，如 RDD、JSON、Hive、Parquet、MySQL、HDFS、S3、JDBC 等。其中 Parquet 是面向分析型业务的列式存储格式，由 Twitter 和 Cloudera 合作开发，2015 年 5 月从 Apache 的孵化器里毕业成为 Apache 的顶级项目。最开始的时候，Spark SQL 用户最喜欢的一个功能是从 Apache Hive 表中读取数据，和从 Parquet 中读取数据一样。从图 10-2 中我们可以看出，Spark SQL 的数据源很丰富，包括 Hive、Avro、Parquet、ORC、JSON、JDBC。Spark SQL 支持一些行业标准化的工具，如 JDBC、ODBC。

Spark SQL 已经是 Spark 阵营中的一个主要组件，为 Spark 提供了查询结构化数据的能力，主打交互查询场景，查询时既可以使用 SQL，也可以使用人们熟知的 DataFrame API（RDD）。Spark SQL 的引擎优化改变了 Hive 负责的引擎。Spark SQL 支持多语言编程，包括 Java、Scala、Python 及 R，开发人员可以根据自己的喜好进行选择，其 API 不仅易于使用，而且功能强大。

10.3 Spark SQL 之 DataFrame

10.3.1 什么是 DataFrame

DataFrame 是一种以 RDD 为基础的分布式数据集合，该数据集合以命名列的方式进行整合。DataFrame 可以理解为关系数据库中的一张表，也可以理解为 R/Python 中的一个数据结构。DataFrame 可以通过多种数据构造，例如结构化的数据文件、Hive 中的表、外部数据库、Spark 计算过程中生成的 RDD 等。DataFrame 与 RDD 的主要区别在于，前者带有 Schema 元信息，即 DataFrame 所表示的二维表数据集的每一列都带有名称和类型。这使得 Spark SQL 得以洞察更多的结构信息，从而对藏于 DataFrame 背后的数据源及作用于 DataFrame 之上的变换进行针对性的优化，最终达到大幅提升运行时效率的目标。DataFrame 的 API 支持 4 种语言：Scala、Java、Python、R。

10.3.2 DataFrame 的创建

在 Spark SQL 中，开发者可以非常便捷地将各种内、外部的单机、分布式数据转换为 DataFrame。以下 Python 示例代码充分体现了 Spark SQL 1.3.0 中

DataFrame 数据源的丰富多样和简单易用。

```
# 从Hive中的users表构造DataFrame
users = sqlContext.table("users")

# 加载S3上的JSON文件
logs = sqlContext.load("s3n://path/to/data.json", "json")

# 加载HDFS上的Parquet文件
clicks = sqlContext.load("hdfs://path/to/data.parquet", "parquet")

# 通过JDBC访问MySQL
comments = sqlContext.jdbc("jdbc:mysql://localhost/comments", "user")

# 将普通RDD转变为DataFrame
rdd = sparkContext.textFile("article.txt") \
          .flatMap(lambda line: line.split()) \
          .map(lambda word: (word, 1)) \
          .reduceByKey(lambda a, b: a + b) \
wordCounts=sqlContext.createDataFrame(rdd,["word", "count"])

# 将本地数据容器转变为DataFrame
data = [("Alice", 21), ("Bob", 24)]
people = sqlContext.createDataFrame(data, ["name", "age"])

# 将Pandas DataFrame转变为Spark DataFrame（Python API特有功能）
sparkDF = sqlContext.createDataFrame(pandasDF)
```

可见，从 Hive 表到外部数据源 API 支持的各种数据源（JSON、Parquet、JDBC），再到 RDD 乃至各种本地数据集，都可以被方便快捷地加载、转换为 DataFrame。这些功能也同样存在于 Spark SQL 的 Scala API 和 Java API 中。

基于 JSON 文件创建 DataFrame 的 Scala 示例如下：

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val df = sqlContext.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
```

基于 JSON 文件创建 DataFrame 的 Java 示例如下：

```
JavaSparkContext sc = ...; // An existing JavaSparkContext.
SQLContext sqlContext = new org.apache.spark.sql.SQLContext(sc);

DataFrame df = sqlContext.read().json("examples/src/main/resources/
people.json");

// Displays the content of the DataFrame to stdout
df.show();
```

10.3.3 DataFrame 的使用

和 R、Pandas 类似，Spark DataFrame 也提供了一整套用于操纵数据的 DSL。这些 DSL 在语义上与 SQL 关系查询非常相近（这也是 Spark SQL 能够为 DataFrame 提供无缝支持的重要原因之一）。以下是一组用户数据分析示例：

```
# 创建一个只包含“年轻”用户的DataFrame
young = users.filter(users.age < 21)

# 也可以使用Pandas风格的语法
young = users[users.age < 21]

# 将所有人的年龄加1
young.select(young.name, young.age + 1)

# 统计年轻用户中各性别人数
young.groupBy("gender").count()

# 将所有年轻用户与另一个名为logs的DataFrame连接起来
young.join(logs, logs.userId == users.userId, "left_outer")
```

10.4 Spark SQL 运行过程分析

Spark SQL 对 SQL 语句的处理和关系型数据库对 SQL 语句的处理采用了类似的方法，首先将 SQL 语句进行解析（Parse），然后形成一个 Tree，后续的如绑定、优化等处理过程都是对 Tree 的操作，而操作的方法是采用 Rule，通过模式匹配，对不同类型的节点采用不同的操作。在整个 SQL 语句的处理过程中，Tree 和 Rule 相互配合，完成了解析、绑定（在 Spark SQL 中称为 Analysis）、优化、物理计划等过程，最终生成可以执行的物理计划。Spark SQL 的核心是把已有的 RDD 加上 Schema 信息，注册成类似 SQL 里的“Table”，对其进行 SQL 查询。

SQLContext 是执行 SQL 的上下文对象，首先来看一下它有哪些成员。

- Catalog：一个存储 <TableName, LogicalPlan> 的 Map 结构，查找关系的目录，注册表，注销表，查询表和逻辑计划关系的类。
- SQLParser：Parse 传入的 SQL 来对语法进行分词，构建语法树，返回一个 LogicalPlan。
- Analyzer：LogicalPlan 的语法分析器。
- Optimizer：LogicalPlan 的优化器。
- LogicalPlan：逻辑计划，由 Catalyst 的 TreeNode 组成，可以看到有 3 种语法树。
- SparkPlanner：包含不同策略的优化策略来优化物理执行计划。
- QueryExecution：SQL 执行的环境上下文。

SQLContext 对 SQL 的解析和执行流程如图 10-3 所示。

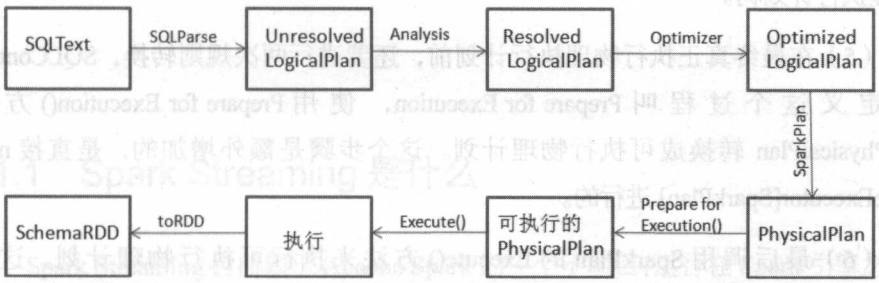


图 10-3　SQLContext 对 SQL 的解析和执行流程

Spark SQL 使用 Catalyst 的树转换框架，分为 4 个阶段：（1）分析逻辑规划（LogicalPlan）来检索引用；（2）优化 LogicalPlan 阶段；（3）物理规划（PhysicalPlan）阶段；（4）Java 字节码查询的编码生成。在 PhysicalPlan 阶段，Catalyst 将生成多个计划，并进行对比。Catalyst 是与 Spark 解耦的一个独立库，是一个 impl-free 的执行计划的生成和优化框架。其他系统如果想基于 Spark 做一些类 SQL、标准 SQL 甚至其他查询语言的查询，则需要基于 Catalyst 提供的解析器、执行计划树结构、逻辑执行计划的处理规则体系等类体系来实现执行计划的解析、生成、优化、映射工作。

主要分为以下几个步骤。

（1）Parse SQL：Spark SQL 以关系计算开始，首先对抽象语法树（AST）或者 DataFrame 词法和语法进行解析，生成 Unresolved LogicalPlan。

（2）Analyzer：使用 Analyzer 结合数据字典（Catalog）进行绑定，生成 Resolved LogicalPlan。SQLContext 内的 Analyzer 由 Catalyst 提供，定义如下：

```
new Analyzer(catalog, EmptyFunctionRegistry, caseSensitive =true)
```

catalog 为 SimpleCatalog 实例，它是用来注册 Table 和查询 Relation 的。而这里的 FunctionRegistry 不支持 lookupFunction() 方法，所以该 Analyzer 不支持 Function 注册，即 UDF。

（3）Optimizer：使用 Optimizer 对 Resolved LogicalPlan 进行优化，生成 Optimized LogicalPlan。Optimizer 里面也定义了几批规则，会按序对执行计划进行优化操作。

（4）优化后的执行计划还要使用 SparkPlan 将 LogicalPlan 转换成 PhysicalPlan。SparkPlan 里面定义了一些策略，目的是根据逻辑执行计划树生成最后可以执行的物理执行计划树。

（5）在最终真正执行物理执行计划前，还要进行两次规则转换，SQLContext 里定义这个过程叫 Prepare for Execution，使用 Prepare for Execution() 方法将 PhysicalPlan 转换成可执行物理计划。这个步骤是额外增加的，是直接 new RuleExecutor[SparkPlan] 进行的。

（6）最后调用 SparkPlan 的 Execute() 方法来执行可执行物理计划。这个 Execute() 方法在每种 SparkPlan 的实现里定义，一般都会递归调用 Children 的 Execute() 方法，所以会触发整棵树的计算。

10.5 小结

本章分析了 Shark 的起源，以及发展到 Saprk SQL 的历程，重点分析了 Spark 的架构和运行过程。Spark SQL 已经成为除 Spark Core 以外最大的 Spark 组件。除了接过 Shark 的“接力棒”，继续为 Spark 用户提供高性能的 SQL on Hadoop 解决方案外，它还为 Spark 带来了通用、高效、多元一体的结构化数据处理能力。

11 Spark Streaming 流数据处理新贵

11.1 Spark Streaming 是什么

Spark Streaming 目前属于 Apache Spark 的一个子项目，运行在 Spark 引擎之上，功能和 Apache Storm 有点类似，专注于实时的流数据处理。

Spark Streaming 支持的数据输入源有很多，例如 Kafka、Flume、Twitter、ZeroMQ 和简单的 TCP 套接字等。数据源获取数据之后，可以使用诸如 map、reduce、join 和 window 等高级函数进行复杂算法的处理。最后还可以将结果保存在很多地方，如 HDFS、数据库等。Spark 数据处理流的输入和输出如图 11-1 所示。

另外，Spark Streaming 也可以和 Spark 的其他子框架，如机器学习 MLlib、图计算 GraphX 等完美结合，适用于特定的场合。

Spark Streaming 是基于 Spark 平台的，因而继承了 Spark 平台的各种优质特性，如很强的容错机制、可扩展性、高吞吐量等。对于 Spark 来说，每个 RDD 都是一个不可变的分布式可重算的数据集，其记录着确定性的操作继承关系（Lineage）。所以只要输入数据是可容错的，那么任意一个 RDD 的分区（Partition）出错或不可用，都是可以利用原始输入数据通过转换操作而重新算出的。



图 11-1 数据处理流的输入和输出

11.2 Spark Streaming 的架构

Spark Streaming 内部处理机制如图 11-2 所示。Spark Streaming 接收到实时输入数据流，将其按照时间片来划分为多批(batch)数据，然后交由 Spark Engine 处理，最终得到处理后的多批结果数据。

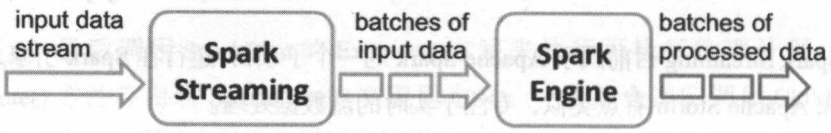


图 11-2 Spark Streaming 内部处理机制

Spark Steaming 中有一个很重要的概念：DStream（Discretized Stream，离散流）。DStream 是 Spark Streaming 的基础抽象，它代表持续性的数据流。这些数据流既可以基于 Kafka、TCP Socket、Flume 等输入数据流来创建，也可以通过现有的 DStream 的 Transformation 操作来获得。从内部实现机制来看，DStream 实际上是由一组 RDD 序列组成的，每个 RDD 都包含了自己特定时间间隔内的数据流，每个 DStream 都包含了一个时间间隔内的数据项的集合，我们可以理解为指定时间间隔内的一个 batch，每个 batch 构成一个 RDD 数据集，所以 DStream 就是一个个 batch 的有序序列，时间是连续的，按照时间间隔将数据流分割成一个个离散的 RDD 数据集。如图 11-3 所示，每个 RDD 都包含两秒（设定的区间长度）的数据。在 Spark Streaming 中，最小长度可以设置为 0.5 秒，因此处理延迟可以达到 1 秒以下。

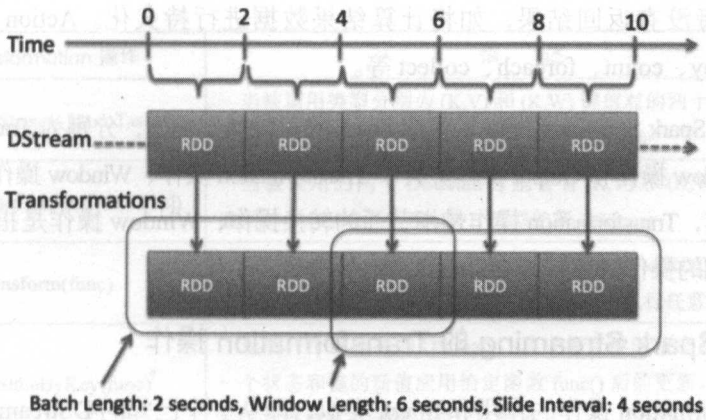


图 11-3 DStream 中的 RDD 序列

Spark Streaming 首先将流数据化整为零，以时间片为单位进行分批，将流处理转换为时间片数据的批处理。随着持续时间的推移，这些处理结果就形成了对应的结果数据流。在此过程中，会涉及图 11-3 中体现的几个因素。

（1）批处理时间间隔（Batch Length）：这是人为地对流数据进行定量的标准，以时间片作为拆分流数据的依据。一个时间片的数据对应一个 RDD 实例。图 11-3 中为 2 秒。

（2）窗口长度（Window Length）：一个窗口覆盖的流数据的时间长度。其必须是批处理时间间隔的倍数。图 11-3 中为 6 秒。

（3）滑动时间间隔（Slide Interval）：前一个窗口到后一个窗口所经过的时间长度。其必须是批处理时间间隔的倍数。图 11-3 中为 4 秒。

11.3 Spark Streaming 的操作

前面我们已经详细讨论过，Spark 支持两种类型的操作：Transformation 和 Action。让我们大概温习一下：Transformation 从一个已知的 RDD 数据集经过转换得到一个新的 RDD 数据集，这些 Transformation 操作包括 map、filter、flatMap、union、join 等。而且 Transformation 具有惰性，调用这些操作并没有立刻执行对已知 RDD 数据集的计算操作，而是等到调用另一类型的 Action 操作才会真正地执行。Action 执行会真正地对 RDD 数据集进行操作，返回一个计算结果给 Driver

程序，或者没有返回结果，如将计算结果数据进行持久化。Action 操作包括 reduceByKey、count、foreach、collect 等。

同样，Spark Streaming 提供了类似 Spark 的三种操作类型，分别为 Transformation 操作、Window 操作和 Output 操作。其中 Transformation 操作、Window 操作可以都归为转换操作，Transformation 操作是指普通的转换操作，Window 操作是指窗口转换操作，它们的操作对象都是 DStream。

11.3.1 Spark Streaming 的 Transformation 操作

Transformation 操作的作用和 Spark 类似，即从一个已知的 DStream 经过转换得到一个新的 DStream。部分 Transformation 操作如表 11-1 所示。

表 11-1 部分 Transformation 操作

Transformation 操作	描　述
map(func)	源 DStream 的每个元素通过函数 func() 返回一个新的 DStream
flatMap(func)	类似于 map 操作，不同的是每个输入元素可以被映射出 0 或者更多的输出元素
filter(func)	在源 DStream 上选择 func() 函数返回仅为 true 的元素，最终返回一个新的 DStream
repartition(numPartitions)	通过输入参数 numPartitions 的值来改变 DStream 的分区大小
union(otherStream)	返回一个包含源 DStream 与其他 DStream 的元素合并后的新 DStream
count()	对源 DStream 内部所有 RDD 的元素数量进行计数，返回一个内部所包含的 RDD 只有一个元素的新 DStream
reduce(func)	使用函数 func()（有两个参数并返回一个结果）将源 DStream 中每个 RDD 的元素进行聚合操作，返回一个内部所包含的 RDD 只有一个元素的新 DStream
countByValue()	计算 DStream 中每个 RDD 内的元素出现的频次并返回新的 DStream[(K, Long)]，其中 K 是 RDD 中元素的类型，Long 是元素出现的频次
reduceByKey(func, [numTasks])	当一个类型为 (K,V) 键值对的 DStream 被调用的时候，返回类型为 (K,V) 键值对的新 DStream，其中每个键的值 V 都是使用聚合函数 func() 汇总。注意：在默认情况下，使用 Spark 的默认并行度提交任务（本地模式下并行度为 2，集群模式下并行度为 8），可以通过配置 numTasks 设置不同的并行任务数

续表

Transformation 操作	描　述
join(otherStream, [numTasks])	当被调用类型分别为 (K,V) 和 (K,W) 键值对的两个 DStream 时，返回类型为 (K, (V,W)) 键值对的一个新 DStream
cogroup(otherStream, [numTasks])	当被调用的两个 DStream 分别含有 (K,V) 和 (K,W) 键值对时，返回一个 (K,Seq[V], Seq[W]) 类型的新 DStream
transform(func)	通过对源 DStream 的每个 RDD 应用 RDD-to-RDD 函数返回一个新的 DStream，这可以用来在 DStream 中执行任意 RDD 操作
updateStateByKey(func)	返回一个新状态的 DStream，其中每个键的状态是根据键的前一个状态和键的新值应用给定函数 func() 后的更新。该方法可以被用来维持每个键的任何状态数据

11.3.2 Spark Streaming 的 Window 操作

Window 操作有助于更有效率地在一组 RDD（a rolling window of time）上进行计算，但是可以更灵活地控制 DStream 的大小（时间间隔大小、数据元素个数），例如 window(windowLength, slideInterval)、countByWindow(windowLength, slideInterval)、reduceByWindow(func, windowLength, slideInterval) 等。部分 Window 操作如表 11-2 所示。

表 11-2　部分 Window 操作

Window 操作	描　述
window(windowLength, slideInterval)	返回一个基于源 DStream 的窗口，批次计算后得到新的 DStream
countByWindow(windowLength,slide Interval)	返回基于滑动窗口的 DStream 中的元素数量
reduceByWindow(func, windowLength, slideInterval)	基于滑动窗口对源 DStream 中的元素进行聚合操作，得到一个新的 DStream
reduceByKeyAndWindow(func,window Length, slideInterval, [numTasks])	基于滑动窗口对 (K,V) 键值对类型的 DStream 中的值按 K 使用聚合函数 func() 进行聚合操作，得到一个新的 DStream
reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])	一个更高效的 reduceByKkeyAndWindow() 的实现版本，先对滑动窗口中新的时间间隔内的数据增量聚合并移去最早的与新增数据量的时间间隔内的数据统计量。例如，计算 *t*+4 秒这个时刻过去 5 秒窗口的 WordCount，那么我们可以将 *t*+3 时刻过去 5 秒的统计量加上 [t+3, t+4] 的统计量，再减去 [t-2, t-1] 的统计量。这种方法可以复用中间三秒的统计量，从而提高统计的效率

续表

Window 操作	描　述
countByValueAndWindow(windowLength, slideInterval, [numTasks])	基于滑动窗口计算源 DStream 中每个 RDD 内每个元素出现的频次并返回 DStream[(K, Long)]，其中 K 是 RDD 中元素的类型，Long 是元素出现的频次。与 countByValue 一样，Reduce 任务的数量可以通过一个可选参数进行配置

11.3.3 Spark Streaming 的 Output 操作

Spark Streaming 的 Output 操作允许我们将 DStream 数据输出到一个外部的存储系统，如数据库或文件系统等。执行 Output 操作类似于执行 Spark 的 Action 操作，使得该操作之前的 Transformation 操作序列真正地执行。部分 Output 操作如表 11-3 所示。

表 11-3　部分 Output 操作

Output 操作	描　述
print()	在 Driver 中打印出 DStream 中数据的前 10 个元素
saveAsTextFiles(prefix, [suffix])	将 DStream 中的内容以文本的形式保存为文本文件，其中每次批处理间隔内产生的文件以 prefix-TIME_IN_MS[.suffix] 方式命名
saveAsObjectFiles(prefix, [suffix])	将 DStream 中的内容按对象序列化并且以 SequenceFile 格式保存。其中每次批处理间隔内产生的文件以 prefix-TIME_IN_MS[.suffix] 方式命名
saveAsHadoopFiles(prefix, [suffix])	将 DStream 中的内容以文本的形式保存为 Hadoop 文件，其中每次批处理间隔内产生的文件以 prefix-TIME_IN_MS[.suffix] 方式命名
foreachRDD(func)	最基本的输出操作，将 func() 函数应用于 DStream 中的 RDD 上。这个操作会输出数据到外部系统，比如保存 RDD 到文件或者网络数据库等。需要注意的是，func() 函数是在运行该 Streaming 应用的 Driver 进程里执行的

11.4 Spark Streaming 性能调优

性能调优一直是大数据处理中最热门的话题，尤其是对于 Spark Streaming 这

样的实时流处理平台，要求则更高。为了适合不同的场景和不同的应用，Spark Streaming 为用户提供了很多可配置项来最大化地满足用户的自定义平台。这些配置项都被赋予默认值，但是默认值不一定能使系统的性能达到最优。我们需要根据集群中机器的性能、数据量等来调节 Spark Streaming 的参数、Spark 的参数，甚至是系统的参数。

1. 合理的批处理时间间隔

此方法可谓 Spark Streaming 性能调优的首选。为什么呢？我们知道，在构建 StreamingContext 的时候，需要传进一个参数，用于设置 Spark Streaming 批处理的时间间隔，然后每隔此时间间隔会提交一个 Job。在 Spark Streaming 中，Job 之间有可能存在依赖关系，后面的 Job 必须确保前面的 Job 执行结束后才能提交。若前面的 Job 执行的时间超出了批处理时间间隔，那么后面的 Job 就无法按时提交，这样就会进一步拖延接下来的 Job，造成后续 Job 的阻塞。因此，设置一个合理的批处理时间间隔以确保作业能够在这个批处理间隔内结束是必需的。

那么到底设置多少合适呢？答案是没有答案。我们可以先发出一段训练序列去学习这个最优的值，首先将这个时间片设置到较大，如果发现作业能很快执行完毕，则可以逐步减少该值；但也不要过小（比如小于 500ms），否则会因频繁提交作业而给整个 Streaming 带来负担，提交和分发任务的延迟就变得不可接受了。

2. 增加并行化，充分利用资源

在整个大数据处理中，最核心的思想莫过于“并行化”。靠高性能机器串行的时代已经过去，目前几乎所有的大数据技术都有“并行化”的影子。但是如何最大化地应用好并行，却是各大数据处理平台争长论短的话题。首先要增加 Job 的并行度，不能让部分节点运行，剩下的节点闲置。其次，可以从接收端来增加并行度。如果 Spark 与外部系统通信，比如 Kafka，则可以使用 Kafka 缓冲来优化 Spark Streaming 到 Kafka 的写入，达到负载均衡，及时处理数据。在这里，优化意味着在多个任务中共享同一个生产者，该操作可以显著地减少由 Kafka 集群建立的新 TCP 连接数。

3. 减少数据序列化、反序列化的负担

Spark Streaming 默认使用 Java 内置的系列化类，虽然可以处理所有从 java.io.Serializable 类继承来的类，将接收到的数据序列化后存储，以减少内存的使用，

但是序列化和反序列化需要更多的 CPU 时间，从而导致其性能不佳。可以使用 Kryo 序列化类和自定义的序列化接口更高效地使用 CPU。例如，可以使用 Kryo 生成的 Avro-generated Java 类来取代反序列化。这个优化对 Spark Streaming 作业也非常有效。但在使用 Kryo 这样的高速序列化工具时，会增加 CPU 的负荷。

4. 设置合理的 GC 策略

不合理的 GC 行为会给程序带来很大的影响，可能延长 Job 的执行时间，引起一系列不可预料的问题。观察 GC 的运行情况，采用不同的 GC 策略以进一步减小内存回收对 Job 运行的影响。通过将 spark.streaming.unpersist 设置为 true，可以将 Spark Streaming 作业对应的 RDD 设置为持久存储。这样做不仅可以显著地减少 Spark 在 RDD 上的内存使用，也可以改善 GC 行为。

5. 及时清除过期数据

前面讲到 Spark Streaming 会将接收到的数据全部存储到内部可用内存区域中。随着时间的推移，有些数据是不需要的，但是这些数据缓存在内存中，会消耗宝贵的内存资源。我们可以通过设置合理的 spark.cleaner.ttl 时长来及时清理超时的无用数据。这个参数需要小心设置，以免后续操作中所需的数据被超时错误处理。我们还可以配置选项 spark.streaming.unpersist 为 true（默认设置为 true）来更智能地去持久化（unpersist）RDD。该配置使系统找出那些不需要经常保有的 RDD，然后去持久化它们。这样做不仅可以减少 Spark RDD 的内存使用，也可以改善垃圾回收行为。

11.5 小结

本章分析了流数据处理技术 Spark Streaming，阐述了 Spark Streaming 的架构和各种操作。正如 Spark Streaming 最初的目标一样，它通过丰富的 API 和基于内存的高速计算引擎让用户可以结合流式处理、批处理和交互查询等应用。因此，Spark Streaming 适合一些需要历史数据和实时数据结合分析的应用场合。当然，对于实时性要求不是很高的应用也可完全胜任。另外，通过 RDD 的数据重用机制可以得到更高效的容错处理。

12 Spark GraphX 图计算系统

12.1 图计算系统

首先要说明的是，这里的“图”指的不是图画、图片，而是“关系图”。近年来，社交网络越来越发达，我们对社交网络的依赖性越来越强。可以说，通过社交网络，你能找到地球上的任何一个人，这就是“关系图”的威力。

图搞清楚了，那么什么是“图计算”呢？说起“图计算”，可能有许多人还比较陌生，甚至有人会误把它当成专门进行“图像”处理的技术。事实上，我们这里所讲的“关系图计算”和图像处理没有任何关系。“图计算”是以“图论”为基础的对现实世界的一种“图”结构的抽象表达，以及在这种数据结构上的计算模式。

图计算模型由很多个节点（vertex）构成，节点之间通过边（edge）连接，节点和边中都包含了计算状态数据。状态的更新是通过在每个节点上运行一系列的迭代计算来完成的，计算结果是图中所有节点和边的最终状态的聚合。节点计算依赖于自身节点、邻居节点及边的状态，并且可以更新这些状态。通常，在图计算中，基本的数据结构表达如下：

```
G = func(V, E, D)
```

其中，V = vertex（顶点或者节点）；E = edge（边）；D = data（权重）。比如说，对于一个消费者的原始购买行为，有两类节点——用户和产品，边就是购买行为，权重是边上的一个数据结构，可以是购买次数和最后购买时间。对于物理世界的许多数据问题，都可以利用图结构来抽象表达，比如社交网络，网页链接关系，用户传播网络，用户网络点击、浏览和购买行为，甚至消费者评论内容、内容分类标签、产品分类标签等。

很多现实中的计算问题都会涉及大规模的图，经典的例子像网页链接关系和各种社交关系等。这些图的规模（某些情况下，可能达到数十亿的顶点和数万亿的边）使得如何对它们进行高效处理成为一个巨大的挑战。图数据结构很好地表达了数据之间的关联性（dependencies between data）。关联性计算是大数据计算的核心，通过获得数据的关联性，可以从噪声很多的海量数据中抽取有用的信息。例如，通过对购物者之间的关系建模，就能很快找到品位相似的用户，并为之推荐商品。

12.1.1 图存储模式

巨型图的存储总体上有边分割和点分割两种方式。2013 年，GraphLab 2.0 将其存储方式由边分割变为点分割，在性能上取得重大提升，目前基本上被业界广泛接受并使用。

边分割（Edge-Cut）：每个顶点都存储一次，但有的边会被打断分到两台机器上，如图 12-1 所示。这样做的优点是节省存储空间；缺点是对图进行基于边的计算时，对于一条两个顶点被分到不同机器上的边来说，要跨机器通信传输数据，内网通信流量大。

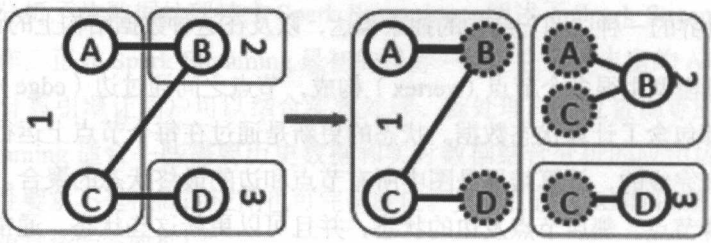


图 12-1　边分割

点分割（Vertex-Cut）：每条边只存储一次，都只会出现在一台机器上，如

图 12-2 所示。这样做的缺点是邻居多的点会被复制到多台机器上，增加了存储开销，同时会引发数据同步问题；优点是可以大幅减少内网通信量。

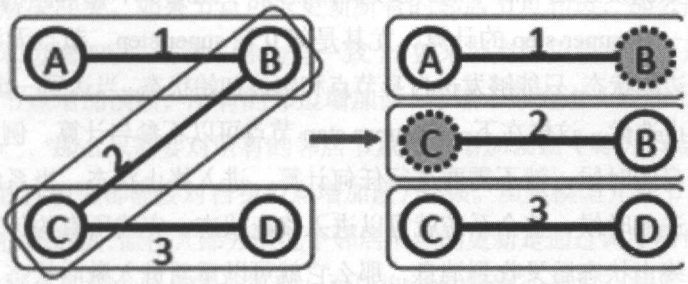


图 12-2 点分割

虽然两种方法各有利弊，但现在以点分割为主，各种分布式图计算框架都将自己底层的存储形式变成了点分割。主要原因有以下两点：

（1）磁盘价格下降，存储空间不再是问题，而内网的通信资源没有突破性进展，集群计算时内网带宽是宝贵的，时间比磁盘更珍贵。这点类似于常见的空间换时间的策略。

（2）在当前的应用场景中，绝大多数网络都是“无尺度网络”，遵循幂律分布，不同点的邻居数量相差悬殊。而边分割会使那些多邻居的点所相连的边大多数被分到不同的机器上，这样的数据分布会使得内网带宽更加捉襟见肘，于是边分割存储方式被渐渐抛弃了。

12.1.2 图计算模式

为应对复杂的处理需求，出现了许多新型的基于图的计算平台和引擎。下面介绍两种具有代表性的图处理模型。

1. Pregel

Pregel 是一个同步图计算模型，整个计算过程由多步迭代构成，每一步称为一个 super-step。在每一个 super-step 中各个节点并行执行计算，在全部节点均完成一次迭代计算之后才进入下一个 super-step。Pregel 模型中的每个节点存储自身状态（初始时还需要存储所有初始邻居节点的 ID），但是并不能直接获得邻居节点及入边的状态数据，这些状态数据是通过消息来传递的。每个节点都能向邻居

节点发送消息，消息中包含自身状态数据及出边的状态数据。节点通过遍历收到的消息来获取所有的依赖状态数据，完成计算，更新自身状态，并将新的状态数据及出边的数据发送给邻居节点。发送的消息并不能用于本轮 super-step 的计算，而是用于下一轮 super-step 的计算，尤其是第 0 轮 super-step，节点无法获取到邻居节点和入边的状态，只能够发送自身节点和出边初始状态。当达到一定的条件后，节点可以终止迭代，这样在下一个 super-step 节点可以不参与计算。例如节点没有收到任何消息的时候，就不需要进行任何计算，进入终止状态。当系统中不再有任何消息发送的时候，整个系统就可以进入终止状态，完成所有的计算。如果某个节点进入终止状态后又收到消息，那么它就可以重新进入激活状态，在下一个 super-step 参与计算。

该模型的优点是图拓扑可以动态变化，节点可以向非邻居节点发送消息，从而建立边的关系。该模型的一个缺点是无法直接获得邻居节点和入边的状态数据，必须依赖于所有邻居发送的消息。在类似于 PageRank 模型的计算中，即使节点不再发生任何状态变更，也必须向邻居节点发送消息，否则邻居节点就无法计算正确的 Rank 值。另一个缺点在于每个 super-step 的同步开销，必须等待所有节点中最慢的一个完成计算后才能开始下一个 super-step。

在模型实现上，多个节点可以放在同一台机器（称为 Worker）上执行，另外还需要一个额外的角色 Master 来协调全局 super-step 的执行。每个 Worker 向 Master 汇报本 Worker 上节点的执行情况，当所有节点执行完毕后，Master 通知各个 Worker 开始下一个 super-step。Master 还可以在每轮 super-step 结束时执行一些聚合操作，计算全局统计信息。实现层面上要考虑的一个重要问题是节点划分，越多相邻的节点放在同一台机器上执行，就可以越多地减少 Worker 之间的消息通信。在容错方面，如果某台 Worker 出现故障，那么需要将该 Worker 上的全部节点调度到新的机器上执行，但是将这些节点恢复到原来的状态需要重新执行全部的 super-step。为了减少这种开销，可以让所有 Worker 定期将状态快照到永久存储（远程），当某个 Worker 发生故障的时候，所有 Worker 载入最近的一个快照开始执行。

2. GraphLab

GraphLab 是一个单机异步图计算模型，模型的数据存储在 DataGraph 中，包括节点的数据和边的数据。每个节点可以直接获得全部邻居节点及入边的状态数据，独立及并行地执行计算，并直接更新自身、邻居和边的状态数据。由于任意

两个节点的计算是并行的，两个节点对同一个节点或者边的读 / 写访问可能会有冲突，此时就需要采用一些方式避免冲突。例如按照不同的访问模式可以使用不同的方法来解决冲突。如果节点可以更新所有的邻居节点和边，那么需要对所有的邻居节点和边增加读 / 写锁（称为全一致）。如果节点只更新邻边，那么需要对所有的邻居节点增加读锁，所有的邻边增加读 / 写锁（称为边一致性）。如果节点仅仅更新自己，那么只需要对所有的邻居节点和边增加读锁（称为节点一致）。当然，任何一致性模型都需要对自身节点增加读 / 写锁。虽然模型允许节点直接更新邻居节点的状态数据，但在大部分情况下邻居节点的更新是通过调度器（Scheduler）来完成的。节点在完成自身的计算后，可以向调度器发送请求让邻居节点执行更新计算，调度器按照不同的设置（FIFO，优先级）动态决定下一个要更新的节点。当然也有静态的调度器，这种调度顺序是静态决定好的，例如 Set 调度器。Set 调度器将图的节点划分为多个不冲突的子集，子集内的节点可以并行执行，当某个子集内的所有节点全部执行完毕后才开始调度下一个子集内的节点执行。

模型中还包含一张全局共享的数据表（SDT），里面的数据也可以被每个节点读取，但是只能被全局方法 Sync 所更新。Sync 方法周期性地按照固定顺序聚合每个节点的状态数据，并更新全局数据表。当全局数据表的目标值收敛到一定程度后，或者调度器内再也没有任何任务的时候，整个计算就可以结束了。由于 GraphLab 是单机模型，实现上 DataGraph 和 SDT 可以存储在内存中，由各个进程共享访问。

Distributed GraphLab 是 GraphLab 的分布式版本，和单机版本的最大区别在于模型的数据存储在分布式环境下。这就造成了如下几个问题：（1）需要对节点和边进行良好的划分，以便尽可能地减少跨机器间的数据传输；（2）邻居节点和邻边的状态数据的获取和一致性维护；（3）需要考虑机器的故障恢复。问题（1）本身是一个很复杂的问题，并不能很容易地解决。简单一点的办法是将图划分为很多个子图，然后将这些子图均匀地分布到集群中去。子图不仅包括自身的节点和边，还包括不在子图内但是和子图内的节点有连接的节点和边，这些节点和边称为 Ghost。当将一台机器上的子图合并时，就可以识别出不在此机器上的节点。这些节点缓存了远程机器上的节点和边的状态数据，可以减少计算时的数据传输；同时为 Ghost 数据引入版本概念，解决分布式数据更新的问题。解决问题（2）需要引入分布式锁，这些锁仅在机器上的节点访问到 Ghost 节点的时候才向远程节点申请读 / 写锁。和 Pregel 一样，Distributed GraphLab 也需要定期进行分布式快

照来备份所有的状态数据到分布式存储系统上。

这个模型的优点是没有同步开销，节点执行完一次迭代后立即开始下一次迭代。而对于某些需要很多轮迭代才能收敛的局部节点，通过动态优先级调度可以加快这些节点的收敛速度，进而提升整体的收敛速度。另外，由于可以直接访问邻居节点及邻边的状态数据，因此并不需要发送冗余的消息以保证邻居节点的计算（相对于 Pregel 来说）。但这同时也是一个缺点，在分布式环境下邻居节点的计算并不是一件容易的事。

12.2 Spark GraphX 的框架

在设计 GraphX 时，点分割和 GAS 都已成熟，并在设计和编码中针对它们进行了优化，在功能和性能之间寻找最佳的平衡点。如同 Spark 本身，每个子模块都有一个核心抽象，GraphX 的核心抽象是 Resilient Distributed Property Graph，一种点和边都带属性的有向多重图。它扩展了 Spark RDD 的抽象，有 Table 和 Graph 两种视图，而只需要一份物理存储。两种视图都有自己独有的操作符，从而获得了灵活操作和执行效率。

如同 Spark，GraphX 的代码非常简洁。GraphX 的核心代码只有 3000 多行，而在此之上实现的 Pregel 模型只要短短的 20 多行。GraphX 的代码结构整体如图 12-3 所示，其中大部分的实现是围绕 Partition 的优化进行的，这在某种程度上说明了点分割的存储和相应的计算优化的确是图计算框架的重点和难点。

GraphX 的底层设计有以下几个关键点：

（1）对 Graph 视图的所有操作最终都会转换成其关联的 Table 视图的 RDD 操作来完成。这样对一张图的计算，最终在逻辑上等价于一系列 RDD 的转换过程。因此，Graph 最终具备了 RDD 的 3 个关键特性：Immutable、Distributed 和 Fault-Tolerant。其中最关键的是 Immutable（不变性）。逻辑上，所有图的转换和操作都产生了一张新图；物理上，GraphX 会有一定程度的不变顶点和边的复用优化，对用户透明。

（2）两种视图底层共用的物理数据，由 RDD[Vertex-Partition] 和 RDD[EdgePartition] 两个 RDD 组成。点和边实际都不是以表 Collection[tuple] 的形式存储的，而是由

VertexPartition/EdgePartition 在内部存储一个带索引结构的分片数据块，以加快不同视图下的遍历速度。不变的索引结构在 RDD 转换过程中是共用的，降低了计算和存储开销。

（3）图的分布式存储采用点分割模式，而且使用 partitionBy 方法，由用户指定不同的划分策略（PartitionStrategy）。划分策略会将边分配到各个 EdgePartition，将顶点 Master 分配到各个 VertexPartition，EdgePartition 也会缓存本地边关联点的 Ghost 副本。划分策略的不同会影响到需要缓存的 Ghost 副本数量，以及每个 EdgePartition 分配的边的均衡程度，因而需要根据图的结构特征选取最佳策略。目前有 EdgePartition2d、EdgePartition1d、RandomVertexCut 和 CanonicalRandomVertexCut 4 种策略。在淘宝大部分场景下，EdgePartition2d 策略效果最好。

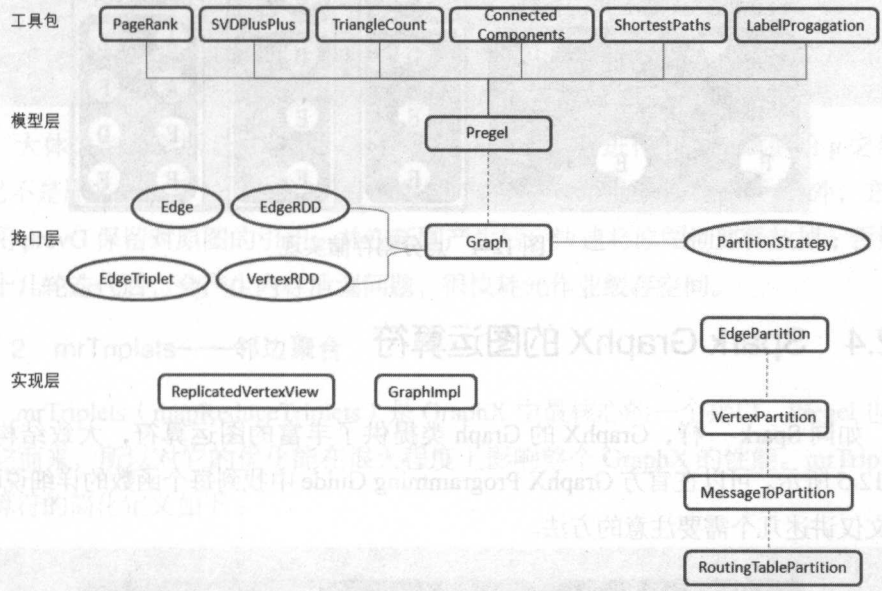


图 12-3　GraphX 代码结构

12.3　Spark GraphX 的存储模式

GraphX 借鉴 PowerGraph，采用的是 Vertex-Cut（点分割）方式存储图，用三个 RDD 存储图数据信息。

（1）VertexTable(id, data)：id 为 Vertex id，data 为 Edge data。

（2）EdgeTable(pid, src, dst, data)：pid 为 Partion id，src 为源顶点 id，dst 为目的顶点 id。

（3）RoutingTable(id, pid)：id 为 Vertex id，pid 为 Partion id。

点分割存储实现如图 12-4 所示。

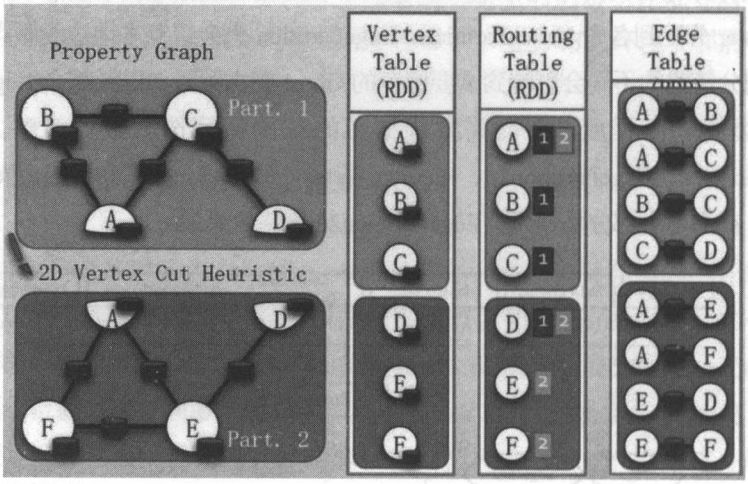


图 12-4　点分割存储实现

12.4　Spark GraphX 的图运算符

如同 Spark 一样，GraphX 的 Graph 类提供了丰富的图运算符，大致结构如图 12-5 所示。可以在官方 GraphX Programming Guide 中找到每个函数的详细说明，本文仅讲述几个需要注意的方法。

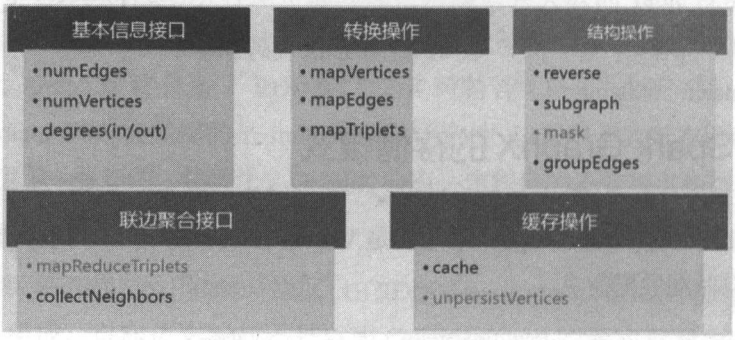


图 12-5　Graph 类提供的图运算符

1. 图的 cache

每张图由 3 个 RDD 组成，所以会占用更多的内存。相应图的 cache、unpersist 和 checkpoint，更需要注意使用技巧。出于最大限度复用边的理念，GraphX 的默认接口只提供了 unpersistVertices() 方法。如果要释放边，则需调用 g.edges.unpersist() 方法才行，这给用户带来了一定的不便，但为 GraphX 的优化提供了便利和空间。参考 GraphX 的 Pregel 代码，对于一张大图，目前最佳的实践如下：

```
var g=...
var prevG:Graph[VD,ED]=null
while(...){
preG = g
g = doSomething(g)
g.cache()
prevG.unpersistVertices(blocking=false)
prevG.edges.unpersist(blocking=false)
}
```

大体之意是根据 GraphX 中 Graph 的不变性，对 g 进行操作并赋值给 g 之后，g 已不是原来的 g 了，而且会在下一轮迭代中使用，所以必须 cache。另外，必须先用 prevG 保留对原图的引用，并在新图产生后，快速将原图彻底释放掉；否则，在十几轮迭代后，会产生内存泄漏问题，很快耗光作业缓存空间。

2. mrTriplets——邻边聚合

mrTriplets（mapReduceTriplets）是 GraphX 中最核心的一个接口，Pregel 也基于它而来，所以对它的优化能在很大程度上影响整个 GraphX 的性能。mrTriplets 运算符的简化定义如下：

```
def mapReduceTriplets[A](
map: EdgeTriplet[VD,ED] =>
Iterator[(VertexID,A)],
      reduce(A,A) => A)
:VertexRDD[A]
```

它的计算过程为：map，应用于每个 Triplet 上，生成一个或者多个消息，消息以 Triplet 关联的两个顶点中的任意一个或两个为目标顶点；reduce，应用于每个 Vertex 上，将发送给每个顶点的消息合并起来。

mrTriplets 最后返回的是一个 VertexRDD[A]，包含每个顶点聚合之后的消息（类

型为 A），没有接收到消息的顶点不会包含在返回的 VertexRDD 中。

在最近的版本中，GraphX 针对 mrTriplets 进行了一些优化，对于 Pregel 及所有上层算法工具包的性能都有重大影响，主要包括以下几点。

（1）Caching for Iterative mrTriplets & Incremental Updates for Iterative mrTriplets：在很多图分析算法中，不同点的收敛速度变化很大。在迭代后期，只有很少的点会有更新。因此，对于没有更新的点，在下一次 mrTriplets 计算时 EdgeRDD 无须更新相应点值的本地缓存，从而大幅降低了通信开销。

（2）Indexing Active Edges：没有更新的顶点在下一轮迭代时不需要向邻居重新发送消息。因此，mrTriplets 在遍历边时，如果一条边的邻居点值在上一轮迭代时没有更新，则直接跳过，从而避免了大量无用的计算和通信。

（3）Join Elimination：Triplet 是由一条边和两个邻居点组成的三元组，操作 Triplet 的 map 函数常常只需访问其两个邻居点值中的一个。例如，在 PageRank 计算中，一个点值的更新只与其源顶点的值有关，而与其所指向的目的顶点的值无关。那么在 mrTriplets 计算中，就不需要 VertexRDD 和 EdgeRDD 的 3-way join，而只需要 2-way join。

所有这些优化使 GraphX 的性能逐渐逼近 GraphLab。虽然还有一定差距，但一体化的流水线服务和丰富的编程接口可以弥补性能的微小差距。

3. 进化的 Pregel 模型

GraphX 中的 Pregel 接口并不严格遵循 Pregel 模型，它是一个参考 GAS 改进的 Pregel 模型。其定义如下：

```
def pregel[A](initialMsg:A,maxIterations:
    Int, activeDirection:EdgeDirection)(
    vprog:(VertexID,VD,A)=>VD,
sendMsg:EdgeTriplet[VD,ED]=>
    Iterator[(VertexID,A)],
    mergeMsg:(A,A)=>A)
    :Graph[VD,ED]
```

这种基于 mrTrilets() 方法的 Pregel 模型与标准 Pregel 的最大区别是，它的第二段参数体接收的是 3 个函数参数，而不接收 messageList。它不会在单个顶点上进行消息遍历，而是将顶点的多个 Ghost 副本收到的消息聚合后发送给 Master 副本，

再使用 vprog() 函数来更新点值。消息的接收和发送都被自动并行化处理，无须担心超级节点的问题。

常见的代码模板如下：

```
//更新顶点
Vprog(vid:Long,vert:Vertex,msg:Double):
Vertex = {
      v.score = msg + (1 - ALPHA) *v.weight
}
//发送消息
SendMsg(edgeTriplet:EdgeTriplet[…]):
Iterator[(Long.Double)]
      (destId,ALPHA * edgeTriplet.srcAttr.
Score * edgeTriplet.attr.weight)
}
//合并消息
mergeMsg(v1:Double,v2: Double):Double = {
}
```

可以看到 GraphX 设计这个模型的用意。它综合了 Pregel 和 GAS 的优点，即接口相对简单，同时又保证性能，可以应对点分割的图存储模式，胜任符合幂律分布的自然图的大型计算。另外，值得注意的是，官方的 Pregel 版本是最简单的版本。对于复杂的业务场景，根据这个版本扩展一个定制的 Pregel 是很常见的做法。

12.5 小结

本章分析了图计算的存储模式和计算模式，重点阐述了 Spark GraphX。由于天然符合互联网中很多场景的需求，图计算正受到越来越多的青睐。由于 Spark GraphX 性能良好，同时又有丰富的功能和运算符，能在海量数据上自如运行复杂的图算法，所以 Spark GraphX 作为 Spark 技术堆栈中的一员，担负起了 Spark 在图计算领域的重任。

13 Spark Cluster 管理

13.1 Spark Cluster 部署

要使用和学习 Spark，首先要去 Apache Spark 官网下载 Spark 源代码及编译好的安装文件。Apache 官方的下载地址为 http://spark.apache.org/downloads.html。

关于如何安装和配置，非常简单，无须赘述。如果你想搭建一个 Spark Cluster，可以上网搜索，会有详细的教程。

安装完成后，可以通过 start-master.sh 来启动 Spark Master。Spark 提供了一个 Web UI 来查看集群信息并且监控执行结果，默认地址是 http://<spark_master_ip>:8080，如图 13-1 所示。

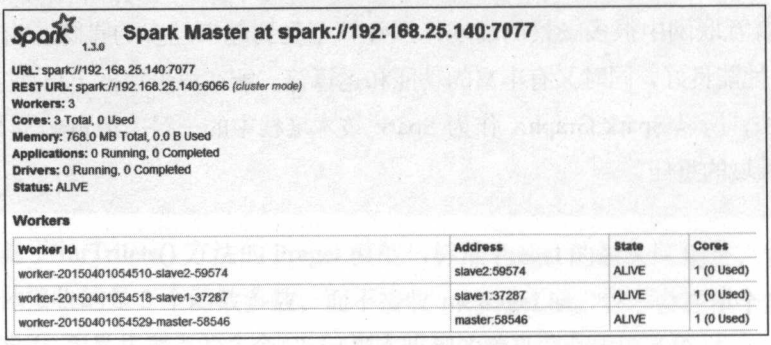


图 13-1　Spark Web UI

在安装过程中需要注意防火墙的问题，特别是 Spark on YARN 模式，有可能在 NodeManager 端的日志里出现类似 Connection Refused 的错误信息，需要检查防火墙是否禁止 80 端口访问。

13.2 Spark Cluster 管理与监控

13.2.1 内存优化机制

Spark 中负责具体计算任务的是 Executor，每个 Executor 上的内存大小是可以配置的。从 Executor 的内存中划出一定的比例用于 RDD 的缓存，其他内存用于 Task 的任务计算，比如保存新创建的对象等。Executor 的内存大小通过 spark.executor.memory 参数配置，默认是 512MB。上述比例通过 spark.storage.memoryFraction 参数配置，默认是 0.6。即默认每个 Executor 的内存大小是 512MB，其中 512MB×0.6=307.2MB 用于 RDD 缓存，其余 512MB×0.4=204.8MB 用于 Task 任务计算。

以下是常遇到的问题及其优化方案：

- 如果 Executor 报 OOM 内存不足，则需要考虑增大 spark.executor.memory。
- 如果频繁 Full GC，则可能是 Executor 中用于 Task 任务计算的内存不足，需要考虑降低 spark.storage.memoryFraction 的比例，即减小用于缓存的内存大小，增大用于 Task 任务计算的内存大小；需要考虑优化 RDD 中的数据结构，以减小数据占用的内存大小。
- 如果频繁 Minor GC，则需要考虑增大年轻代内存的大小。这里有一个问题：如何查看 GC 的情况？可以在 spark-env.sh 中设置 JAVA_OPTS 参数以打印 GC 的相关信息。这样如果有 GC 发生，就可以在 Master 和 Work 的日志上看到。
- 关于内存优化，还有一个比较有效的方法是调整存储级别。默认的存储级别是 MEMORY_ONLY，即以对象的形式只存储在内存中。如果内存不够用，则一种优化方式是将存储级别改为 MEMORYONLYSER。这种级别会在对象进行序列化后再存入内存，以减小占用的内存空间。

13.2.2 Spark 日志系统

Spark 使用 log4j 来记录。用户可以在 conf 目录中添加 log4j.properties 文件来

配置。一种方法是复制本地已存在的 log4j.properties.template，在 Spark 的 conf 目录下，把 log4j.properties.template 修改为 log4j.properties。原来的内容如下：

```
log4j.rootCategory=INFO, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss}
%p %c{1}: %m%n

#Settings to quiet third party logs that are too verbose
log4j.logger.org.spark-project.jetty=WARN
log4j.logger.org.spark-project.jetty.util.component.AbstractLifeCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=INFO
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter=INFO
```

把 log4j.rootCategory=INFO, console 改为log4j.rootCategory=WARN, console 即可抑制 Spark 把 INFO 级别的日志打印到控制台上。如果要显示全面的信息，则把 INFO 改为 DEBUG。

如果希望一方面把代码中的 println 打印到控制台，另一方面又保留 Spark 本身输出的日志，则可以将它输出到日志文件中。

```
log4j.rootCategory=INFO, console,FILE
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd
HH:mm:ss} %p %c{1}: %m%n

# Settings to quiet third party logs that are too verbose
log4j.logger.org.eclipse.jetty=WARN
log4j.logger.org.eclipse.jetty.util.component.AbstractLifeCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=INFO
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter=
INFO

log4j.appender.FILE=org.apache.log4j.DailyRollingFileAppender
log4j.appender.FILE.Threshold=DEBUG
log4j.appender.FILE.file=/home/hadoop/spark.log
log4j.appender.FILE.DatePattern='.'yyyy-MM-dd
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.FILE.layout.ConversionPattern=[%-5p] [%d{yyyy-MM-dd
HH:mm:ss}] [%C{1}:%M:%L] %m%n
# spark
log4j.logger.org.apache.spark=INFO
```

上面的操作中，Spark 的日志一方面打印到控制台，另一方面写入 /home/hadoop/spark.log 中。这是日志的继承特性，后面再来改进，目前把 log4j.rootCategory=INFO, console, FILE 改为 log4j.rootCategory=INFO, FILE 即可。

13.3 Spark 高可用性

HA（High Available）即高可用性，是保证大型系统在遇到突发故障的情况下可以继续运行的主要手段。大数据处理平台对此也有很高的要求，如果一个管理节点突然出现故障，则备用节点可以很快承担起主节点的功能。

接下来我们重点讨论一下 Standalone 模式下的容灾手段。前面已经提到，Spark Standalone 是 Master-Slave 架构的集群模式，和大部分的 Master-Slave 架构集群类似，都存在 Master 单点故障的问题，一旦 Master 出现故障，Spark Cluster 将不能正常工作，所以 Master 节点的容灾至关重要。目前 Spark 提供了两种方式来解决此问题。

1. 文件系统的故障恢复模式

这种方法适合当 Master 进程挂掉之后，直接重启即可。使用 FILESYSTEM 参数指定一个目录，把当前各个节点的状态写入文件系统。例如：

```
spark.deploy.recoveryMode=FILESYSTEM
spark.deploy.recoveryDirectory=/tmp/spark/dataDir
```

当 stop-master.sh 杀死 Master 之后，状态没有及时更新，再次启动的时候，会增加一分钟的启动时间来等待原来的连接超时。recoveryDirectory 最好能够使用一个 NFS，这样当一个 Master 失败之后，就可以启动另一个 Master。

2. 基于 ZooKeeper 的 HA 方案

基于 ZooKeeper 的 HA 方案和 HDFS 的 NameNode 的 HA 方案类似，Active Master 挂掉之后，Standby Master 会立即切换过去继续对外提供服务。同时这种基于 ZooKeeper 的 HA 方案也被很多分布式框架采用，例如 Storm、HBase 等。

Spark 采用 ZooKeeper 作为它的 Active 和 Standby 切换的工具，设置也很简单。一个完整的切换通常需要 1～2 分钟，这时候新提交的作业会受到影响，而之前提交的作业不会受到影响。

在 spark-env.sh 中添加以下设置：

```
spark.deploy.recoveryMode=ZOOKEEPER
spark.deploy.zookeeper.url=192.168.1.1:2181,192.168.1.:2181
spark.deploy.zookeeper.dir=/spark
export SPARK_DAEMON_JAVA_OPTS="-Dspark.deploy.recoveryMode=ZOOKEEPER
-Dspark.deploy.zookeeper.url=hadoop.Master:2181,hadoop.
SlaveT1:2181,hadoop.SlaveT2:2181"
```

这里就有一个问题：集群里有多个 Master，我们在连接的时候，该连接谁？如果有多个 Master，若要实例化 SparkContext，则使用 spark://host1:port1, host2:port2 这样的地址，这样它会同时注册两个 Master，一个失效了，还有另外一个。

如果不愿意配置高可用，只是想失败的时候再恢复一下，那就使用 FILESYSTEM 指定一个目录，把当前各个节点的状态写入文件系统。

```
spark.deploy.recoveryMode=FILESYSTEM
spark.deploy.recoveryDirectory=/usr/lib/spark/dataDir
```

当 stop-master.sh 杀掉 Master 之后，状态没有及时更新，再次启动的时候，会增加一分钟的启动时间来等待原来的连接超时。

recoveryDirectory 最好能够使用一个 NFS，这样在一个 Master 失败之后，就可以启动另一个 Master。

13.4 小结

Spark Cluster 管理和维护是一件非常麻烦的事情，可能会出现各种各样的问题。如果出现问题，最好的办法是查看日志。Cluster 管理员不仅要随时查看监控，还要善于查看监控，并及时分析和解决 Cluster 中的报警（Alert）。在 Spark Cluster 中，HA 也是非常重要的，Spark 的高可用性基本上都偏向于负载均衡模式。

第 3 篇

其他大数据处理技术

14 专为流数据而生的 Storm

14.1 Storm 起因

Storm，翻译成中文是“暴风雨”的意思。在大数据时代，“数据”犹如暴风雨一般涌入各大公司的服务器，公司的日常运营经常会生成 TB 级别的数据。数据来源囊括了互联网装置可以捕获的任何类型的数据，如网站、社交媒体、交易型商业数据及其他商业环境中创建的数据。考虑到数据的生成量，实时处理成了许多机构需要面对的首要挑战。

那么，什么是 Storm 呢？一言以蔽之，“一个免费的、开源的分布式实时计算系统”。按照 Storm 架构师的说法，Storm 对于实时计算的意义类似于 Hadoop 对于批处理的意义。当然，Storm 不仅可以用于实时分析，还可以用于在线机器学习、持续计算、分布式远程调用和 ETL 等领域。Storm 简单易用，支持多种编程语言。Storm 也是少数几个使用 Clojure 编写的开源项目之一。Clojure 是一个在 JVM 平台上运行的动态函数式编程语言。国内几家大型公司也在大规模地使用 Storm，比如百度、淘宝，在 Storm 的官网也能看到它们的 Logo。

Storm 的前身是 BackYype。BackYype 用来帮助商家分析和研究它们的产品在社交网络中的影响力，可以处理历史或实时数据。使用的方法是标准队列和线程（Worker），但是这个方案不太令人满意。首先它并不友好，所有的队列和线程

都必须确保可用，这样相对来说显得冗长。大部分逻辑处理用于收发和序列化 / 反序列化消息，一个应用的某个逻辑将会遍及所有线程。2011 年 7 月，Storm 加入 Twitter，很快便得到了发展和支持。随着 Twitter 平台的发力，越来越多的开发者成为 Storm 的用户。出于对系统性能监控需求的考虑，Storm 提供了相应的指标 API，方便人们把相关指标数据与自己喜好的监控系统进行整合。2013 年，Storm 投入 Apache 的怀抱，并于 2014 年 9 月成为 Apache 官方的顶级项目。

我们知道，任何一个项目的成功都需要真真切切地解决一个实际问题。那么，Storm 如此成功，解决了什么问题呢？

众所周知，在大数据处理方面，很多场合用的是鼎鼎大名的 Hadoop。作为一个批处理系统，Hadoop 以其吞吐量大、自动容错等优点，在海量数据处理方面得到了广泛的应用。但是我们刚才说了，是“很多”场合，而不是“所有”场合，比如需要实时处理的大数据场合，Hadoop 就显得力不从心。因为 Hadoop 设计之初就是为了处理离线的大数据，天然就是为批处理而生的，这也是业界的共识。如果 Hadoop 能独占鳌头，就不会有 Spark、Storm、S4 这些实时的计算系统相继而生了。那么，在实时大数据计算场合，都有哪些要求，或者说需要解决哪些问题呢？

（1）低延迟。因为是实时计算系统，所以延迟必须要低。

（2）高性能。性能不高就是浪费机器。

（3）分布式。系统都是为应用场景而生的，如果你的应用场景、数据和计算单机就能搞定，那就不用考虑这些复杂的问题了。我们所说的是单机搞不定的情况。

（4）可扩展。伴随着业务的发展，我们的数据量、计算量可能会越来越大，所以希望这个系统是可扩展的。

（5）容错。这是分布式系统的通用问题。一个节点出现故障不能影响整个系统的应用。

如果仅仅需要解决这 5 个问题，那么可能会有无数种方案，而且各有千秋。随便举一种方案，如使用消息队列 + 分布在各台机器上的工作进程就可以了。我们再继续往下看。

（1）容易在上面开发应用程序。你设计的系统需要应用程序开发人员考虑各个处理组件的分布、消息的传递吗？如果需要，那就有点麻烦，开发人员可能会用不好，也可能不想用。

（2）消息不丢失。用户发布的消息不能在实时处理的时候丢失了，对吧？更严格一点，如果是一个精确度要求很高的应用，那么它处理的消息要适量才行。这个要求有点高。

（3）消息严格有序。有些消息之间是有强相关性的，比如同一个宝贝的更新和删除操作消息，如果处理时顺序错乱，则效果完全不一样。

鉴于以上需求，Storm 便诞生了。同 Hadoop 一样，Storm 也可以处理大批量的数据，然而 Storm 在保证高可靠性的前提下还可以让处理进行得更加实时；也就是说，所有的消息都会被处理。Storm 同样具备容错和分布计算等特性，这使得 Storm 可以扩展到不同的机器上进行大批量的数据处理。

14.2 Storm 的架构与组件

Storm 集群主要由一个主节点（Master Node）和一组工作节点（Worker Node）组成，通过 ZooKeeper 集群进行协调。

主节点通常运行一个后台程序，名为 Nimbus，用于响应分布在集群中的节点，分配任务和监测故障。这类似于 Hadoop 中的 JobTracker。

工作节点同样会运行一个后台程序，名为 Supervisor，用于收听工作指派并基于要求运行工作进程。每个工作节点都是 Topology（拓扑）中一个子集的实现。

Nimbus 和 Supervisor 之间的协调则通过 ZooKeeper 系统或者集群进行。ZooKeeper 是完成 Supervisor 和 Nimbus 之间协调的服务。而应用程序实现实时的逻辑则被封装进 Storm 中的 Topology。Topology 则是一组由 Spouts（数据源）和 Bolts（数据操作）通过 Stream Groupings 进行连接的图。Storm 的架构和组件如图 14-1 所示。

Storm 中的主要组件及其职责介绍如下。

Topology：Storm 中运行的一个实时应用程序。在 Storm 中，首先要设计一

个用于实时计算的图状结构，也就是拓扑（Topology）。这个拓扑将会被提交给集群，由集群中的主节点（Master Node）分发代码，将任务分配给工作节点（Worker Node）执行。

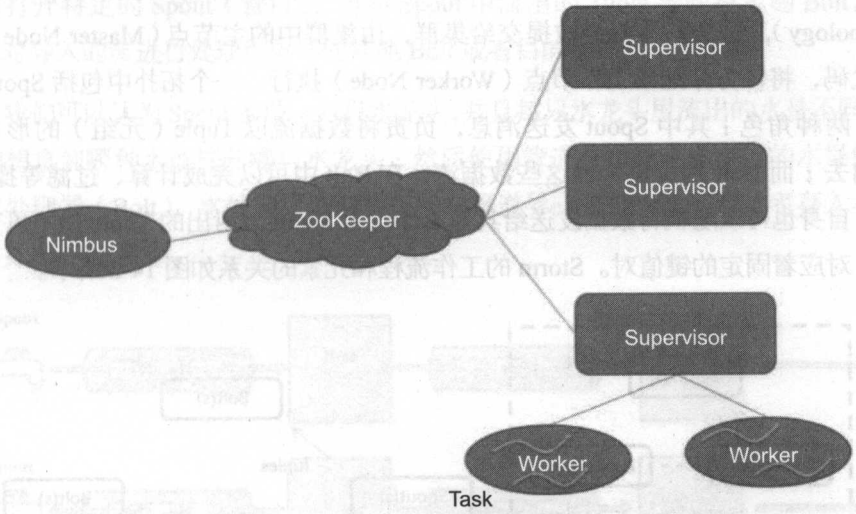


图 14-1 Storm 的架构和组件

Nimbus：主节点上运行的后台程序，负责资源分配和任务调度。

Supervisor：工作节点上运行的后台程序，负责接收 Nimbus 分配的任务，启动和停止属于自己管理的 Worker 进程。

Spout：在一个 Topology 中产生源数据流的组件。简言之，Spout 从来源处读取数据并放入 Topology。Spout 分成可靠和不可靠两种：当 Storm 接收失败时，可靠的 Spout 会对 Tuple（元组，数据项组成的列表）进行重发；而不可靠的 Spout 不会考虑接收成功与否，只发射一次。Spout 中最主要的方法就是 nextTuple()，该方法会发射一个新的 Tuple 到 Topology。如果没有新 Tuple 发射，则会简单地返回。

Bolt：在一个 Topology 中接收数据然后执行处理的组件。Topology 中所有的处理都由 Bolt 完成。Bolt 可以完成任何事，比如连接的过滤、聚合、访问文件 / 数据库等。Bolt 从 Spout 中接收数据并进行处理，如果遇到复杂流的处理，也可能将 Tuple 发送给另一个 Bolt 进行处理。而 Bolt 中最重要的方法是 execute()，以新的 Tuple 作为参数接收。不管是 Spout 还是 Bolt，如果将 Tuple 发射成多个流，这些流都可以通过 declareStream() 方法来声明。

Task：Worker 中每一个 Spout/Bolt 的线程称为一个 Task。

Tuple：一次消息传递的基本单元。

Storm 的工作流程大致为：先设计一个用于实时计算的图状结构，即拓扑（Topology），这个拓扑将会被提交给集群，由集群中的主节点（Master Node）分发代码，将任务分配给工作节点（Worker Node）执行。一个拓扑中包括 Spout 和 Bolt 两种角色：其中 Spout 发送消息，负责将数据流以 Tuple（元组）的形式发送出去；而 Bolt 则负责转换这些数据流，在 Bolt 中可以完成计算、过滤等操作，Bolt 自身也可以随机将数据发送给其他 Bolt。由 Spout 发射出的 Tuple 是不可变数组，对应着固定的键值对。Storm 的工作流程和元素的关系如图 14-2 所示。

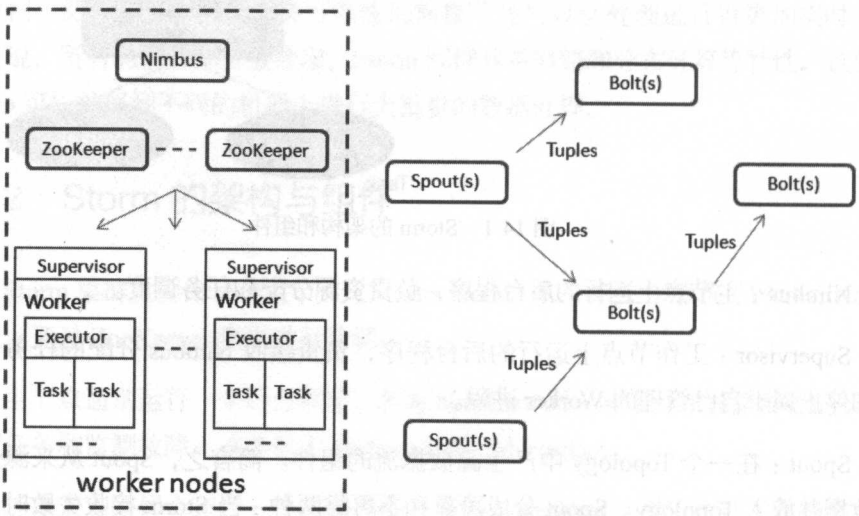


图 14-2　Storm 的工作流程和元素的关系

14.3　Storm 的设计思想

在 Storm 中也有对于流（Stream）的抽象，流是一个不间断的、无界的连续 Tuple。注意 Storm 在建模事件流时，把流中的事件抽象为 Tuple，即元组。

Storm 认为每个 Stream 都有一个 Stream 源，也就是原始元组的源头，所以它将这个源头抽象为 Spout。Spout 可能连接 Twitter API 并不断发出 Tweets，也可能从某个队列中不断读取队列元素并装配为 Tuple 发射。

有了源头（即 Spout）也就有了 Stream，那么该如何处理 Stream 内的 Tuple 呢？同样的思想，Twitter 将流的中间状态抽象为 Bolt，Bolt 可以消费任意数量的输入流，只要将流方向导向该 Bolt，同时它也可以发送新的流给其他 Bolt 使用。这样一来，只需打开特定的 Spout（管口），再将 Spout 中流出的 Tuple 导向特定的 Bolt，由 Bolt 对导入的流进行处理后再导向其他 Bolt 或者目的地。

我们可以认为 Spout 就是一只只水龙头，并且每只水龙头里流出的水是不同的，我们想拿到哪种水就拧开哪只水龙头；然后使用管道将水龙头里流出的水导向一个水处理器（Bolt），水处理器处理后再使用管道导向另一个处理器或者存入容器中，如图 14-3 所示。

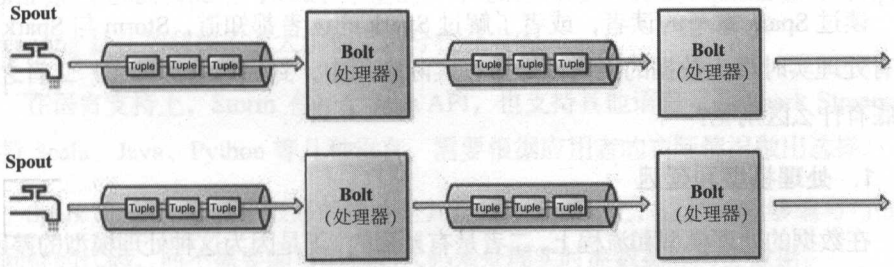


图 14-3　Storm 运行概念图

为了增大水处理效率，我们很自然地想到在同一个水源处接上多只水龙头并使用多个水处理器。Storm 就是这样设计的，来看看图 14-4。

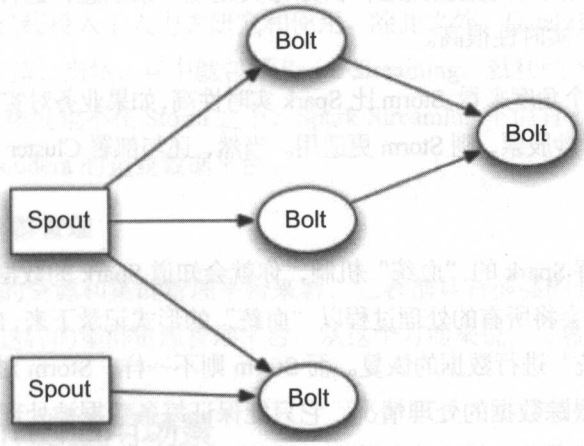


图 14-4　Storm 拓扑

对应上文的介绍，我们可以很容易地理解这张图。这是一张有向无环图，Storm 将这个图抽象为 Topology，即拓扑（的确，拓扑结构是有向无环的）。拓扑是 Storm 中最高层次的一个抽象概念，它可以被提交到 Storm 集群执行。一个拓扑就是一张流转换图，图中的每个节点是一个 Spout 或者 Bolt，图中的边表示 Bolt 订阅了哪些流。当 Spout 或者 Bolt 发送元组到流时，它就发送元组到每个订阅了该流的 Bolt（这就意味着不需要我们手工拉管道，只要预先订阅，Spout 就会将流发送到适当的 Bolt 上）。

14.4 Storm 与 Spark 的区别

读过 Spark 章节的读者，或者了解过 Spark 的读者都知道，Storm 与 Spark 都具有处理实时性大数据的能力；那么在实际应用中，到底该选择谁呢？二者之间到底有什么区别呢？

1. 处理模型和延迟

在数据的处理模型和流程上，二者是有差别的，正是因为这种处理模型的差别，导致处理的延迟也有所不同。

Spark Streaming 的本质其实还是批量处理，只是这个批量是微批量，在短的时间窗口内进行数据实时处理，通常延迟在秒级左右，实时性相对较弱。Storm 可以实现真正流式实时的数据处理，例如每次处理一条消息，这样，延迟就可以控制在秒级以下，实时性很高。

所以，从这个角度来看，Storm 比 Spark 实时性高，如果业务对实时性要求很严格，比如说炒黄金、炒股票，则 Storm 更适用。当然，还与部署 Cluster 的机器性能有关。

2. 容错性

如果你了解 Spark 的“血统”机制，你就会知道 Spark 的数据安全和容错性是天生的。Spark 会将所有的处理过程以“血统”的形式记录下来，出现错误的时候，可以根据“血统”进行数据的恢复。而 Storm 则不一样，Storm 对每条数据进行标记处理，从而跟踪数据的处理情况，它只能保证每条数据被处理一次。但实际情况是，在发生错误的时候，这条数据是被多次处理的，这意味着更新多次的时候可能会导致数据不准确。所以，就容错性而言，虽然 Storm 也是比较出色的，但

是相对于 Spark 来说，还是稍逊一筹。

简言之，如果你需要秒内的延迟，则 Storm 是一个不错的选择，而且没有数据丢失；如果你需要有状态的计算，而且要完全保证每个事件只被处理一次，则 Spark Streaming 更好。Spark Streaming 的编程逻辑可能更容易，因为它类似于批处理程序（Hadoop），特别是在你使用批次（尽管是很小的）时。

所以在容错性这一局，Spark 暂时胜出。

3. 应用和编程 API

Storm 初次是由 Clojure 实现的，而 Spark Streaming 是由 Scala 实现的。Storm 是由 BackType 和 Twitter 开发的，于 2013 年 9 月转到 Apache 旗下；Spark Streaming 是在加利福尼亚大学伯克利分校开发的。

在语言支持上，Storm 有一个 Java API，也支持其他语言；而 Spark Streaming 支持 Scala、Java、Python 等几种语言，需要根据应用者的实际情况做出选择。

Spark Streaming 一个很好的特性是其运行在 Spark 上，这样就能够编写与批处理同样的代码，而不需要编写单独的代码来处理实时流数据和历史数据。

4. 实际应用和成熟度

Storm 已经发布了几年，开源比较早，早在 2011 年，Twitter 就已经开始应用 Storm 来处理业务了。而 Spark Streaming 是一个新的项目，但其发展势头非常迅猛，有很多公司已经投入了人力去研究和使用。除此之外，Databricks 公司专门为 Spark 提供技术支持，当然，其中就包括 Spark Streaming。就社区支持而言，Spark Streaming 的支持热度也不在 Storm 之下，Spark Streaming 不但有 MapReduce 的版本，而且还有 Cloudera 的企业数据平台。

5. 资源和集群管理

从目前支持的资源和集群管理平台来看，二者都具有很强的开放性，都支持 YARN 和 Mesos 这样的集群资源管理平台。从这个方面来说，二者不相上下。

14.5 Storm 的适用场景

分析了这么多 Storm 的思想、架构和优缺点等，那么 Storm 适用于哪些场景呢？

首先是“高度的”实时流数据处理，这对于 Storm 来说是拿手好戏。Storm 的设计初衷就是处理带有突发性的流数据，它可以用来处理源源不断流进来的消息，之后将结果写入某个存储中。所以 Storm 是最佳的流式计算框架，用于处理高速、大型数据流的分布式实时计算系统，在动态处理大量生成的“小数据块”方面性能更优。Storm 这种高可扩展性、能处理高频数据和大规模数据的实时流计算解决方案将被应用于实时搜索、高频交易和社交网络上。而流计算并不是最近的热点，金融机构的交易系统正是一个典型的流计算处理系统，它对系统的实时性和一致性有很高的要求。Storm 可以用来实时处理新数据和更新数据库，兼具容错性和可扩展性。

其次是分布式远程过程调用场合（Distributed Remote Procedure Call）。由于 Storm 的处理组件是分布式的，而且处理延迟极低，所以可以作为一个通用的分布式 RPC 框架来使用。当然，其实我们的搜索引擎本身也是一个分布式 RPC 系统。分布式 RPC 用于对 Storm 上大量的函数调用进行并行计算。对于每一次函数调用，Storm 集群上运行的拓扑接收调用函数的参数信息作为输入流，并将计算结果作为输出流输出。Storm 可以用来并行处理密集查询。Storm 的拓扑结构是一个等待调用信息的分布函数，当它收到一条调用信息后，会对查询进行计算，并返回查询结果。

最后，Storm 适合应用于连续计算（Continuous Computation）场合。Storm 可以进行连续查询并把结果即时反馈给客户，比如将 Twitter 上的热门话题发送到客户端。

总而言之，Storm 在 Hadoop 的基础上提供了实时运算的特性，可以实时处理大数据流。不同于 Hadoop 和 Spark，Storm 不进行数据的收集和存储，它直接通过网络实时地接收数据并进行处理，然后直接通过网络实时传回结果，常用于在线的实时大数据处理。Spark 流模块先汇聚批量数据再进行数据块分发（视作不可变数据进行处理）；而 Storm 只要接收到数据就实时处理并分发。

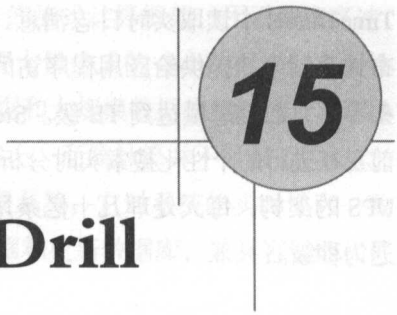
14.6 Storm 的应用

由于 Storm 的内核是 Clojure 编写的（不过大部分的拓展工作都是 Java 编写的），为我们理解它的实现带来了一定的困难，好在大部分情况下 Storm 比较稳定，当然我们也在尽力熟悉 Clojure 的世界。我们在使用 Storm 时通常选择 Java 语言开发应用程序。

在淘宝上，Storm 被广泛用来进行实时日志处理，适用于实时统计、实时风控、实时推荐等场景。一般来说，我们从类 Kafka 的 MetaQ 或者基于 HBase 的 TimeTunnel 中读取实时日志消息，经过一系列处理，最终将处理结果写入一个分布式存储中，提供给应用程序访问。淘宝每天的实时消息量从几百万条到几十亿条不等，数据总量达到 TB 级。Storm 往往会配合分布式存储服务一起使用。在目前正在进行的个性化搜索实时分析项目中，就使用了 TimeTunnel + HBase + Storm + UPS 的架构，每天处理几十亿条用户日志信息，从用户行为发生到完成分析的延迟为秒级。

14.7 小结

本章分析了 Storm 的起因、架构及设计思想，对比了 Storm 和 Spark 的区别，并分析了 Storm 的适用场景。Storm 与其他大数据解决方案的不同之处在于它的处理方式。Hadoop 在本质上是一个批处理系统，数据被引入 Hadoop 文件系统（HDFS）并分发到各个节点进行处理；处理完成后，结果数据返回到 HDFS 供开发者使用。Storm 支持创建拓扑结构来转换没有终点的数据流。不同于 Hadoop 作业，这些转换从不停止，它们会持续处理到达的数据。Storm 没有包括任何存储概念，这就使得 Storm 能够用在各种各样的上下文中，无论数据是从一个非传统来源动态传入，还是存储在数据库等存储系统中，或者由一个控制器对其他设备（比如一个交易系统）进行实时操作。



15 Dremel 和 Drill

15.1 Dremel 和 Drill 的历史背景

首先说明一下我们为什么要把 Google Dremel 和 Apache Drill 拉在一起来讨论，原因是 Drill 是 Apache 推出的 Dremel 的开源版本，所以二者产生的历史背景和想要解决的问题是一致的。

之前我们已经讲过几个大数据的处理产品，例如，如日中天的 Hadoop 和雨后春笋的 Spark 等。那么 Dremel 和 Drill 又是怎么产生的呢？是来解决什么问题的呢？“存在即合理”，每个产品的立足之地就在于解决一个实际的问题。大数据面临的一个很大问题是大多数分析查询都很缓慢且非交互式。Dremel 解决的问题就是大数据下的即时查询，也就是说，在庞大的数据中，快速检索到用户想要的数据。据 Google 的论文数据显示，你能够在几秒钟之内查询多达数 PB 的数据（1PB 等于 100 万 GB）。Dremel 是“交互式”数据分析系统，类似 SQL 风格的语言，让你能够在不编程的前提下轻松地定义特定查询或反复查询，只需把查询命令输入命令行即可。但是与 SQL 的不同之处在于，Dremel 能以极快的速度处理大规模的数据。

缘起缘落，Dremel 缘起何处呢？要从 Google 一篇高大上的论文说起。什么？又是 Google 的论文？是的，不必吃惊，Google 在其他方面的造诣笔者不是很了解，

但是在大数据方面，有很大一部分的思想闪光点均来自 Google 的论文。借此，我们可以大概梳理一下 Google 在大数据方面的思想贡献。

在 2003 年年底和 2004 年，Google 发表了两篇研究论文，其中一篇介绍了 Google File System，它是一个可扩展的分布式文件系统，用于大型的、分布式的、对大量数据进行访问的应用。它运行于廉价的普通计算机服务器上，但可以提供容错功能，并且可以给大量的用户提供总体性能较高的服务。另一篇介绍的是 MapReduce，这是一种编程模型，用于大规模数据集（大于 1TB）的并行运算，能够极大地方便编程人员在不会分布式并行编程的情况下将自己的程序运行在分布式系统上。这就是 Hadoop 的思想精华所在。8 年之后，Hadoop 在网络上得到了广泛的使用，应用领域涉及从数据分析到各种各样的数值计算任务。

但 Google 并没有就此停歇，它试图研发出更好的技术。随后，Google 陆续发表了三篇引人注目的文章：其中一篇详述了 Caffeine，一个为 Google 网络搜索引擎提供支持的软件平台；一篇是关于 Pregel 的，一个用于映射大量在线信息之间关系的“图形数据库”；但最吸引人的还是一篇介绍 Dremel 工具的论文 *Dremel: Interactive Analysis of WebScaleDatasets*。

Dremel 是一种分析信息的方式，它可跨越数千台服务器运行，允许“查询”大量的数据，如 Web 文档集合或数字图书馆，甚至是数以百万计的垃圾信息的数据描述。这类似于使用结构化查询语言分析传统关系数据库，这种方式在过去的几十年里被广泛用于世界各地。专注于大型数据中心规模软件平台的加利福尼亚大学伯克利分校的计算机科学教授 Armando Fox 表示，“如果你事先告诉我 Dremel 可以做什么，那么我不会相信你可以把它开发出来”。

我们知道 Hadoop 擅长“批处理”的大数据处理，目前 Hadoop 已经提供了在庞大数据集上运行类似 SQL 的查询工具（如 Hadoop 生态圈中的项目 Pig 和 Hive）。但它会有一些延迟，例如在部署任务时，可能需要几分钟或者几小时来执行任务，虽然可以得到查询结果，但相比于 Pig 和 Hive，Dremel 几乎是瞬时的。Dremel 可以执行多种查询，而同样的任务如果使用 MapReduce 来执行通常需要一个工作序列，但执行时间却是前者的一小部分。Dremel 可在大约 3 秒钟内处理 1PB 的数据查询请求。

仅仅因为这个功能和优点，就足以让 Dremel 在大数据处理界立足。

Google Dremel 自提出之后，得到了很大的反响，Google 内部也不断地完善它。但由于并非开源，使得其发展和需求的契合受到了一定的阻碍。随后 Apache 基金会支持研发了一个类似 Dremel 的开源版本——Drill。Apache Drill 是一个低延迟的分布式海量数据（涵盖结构化、半结构化及嵌套数据）交互式查询引擎，使用 ANSI SQL 兼容语法，支持本地文件、HDFS、HBase、MongoDB 等后端存储，支持 Parquet、JSON、CSV、TSV、PSV 等数据格式。和 Google Dremel 类似，Drill 满足上千节点的 PB 级别数据的交互式商业智能分析场景。

15.2 Dremel 的原理与应用

Dremel 的技术亮点主要有两个：一是实现了嵌套型（nested）数据的列存储；二是使用了多层查询树，使得任务可以在数千个节点上并行执行和聚合结果。列存储在关系型数据库中并不陌生，它可以减少查询时处理的数据量，有效地提升查询效率。Dremel 的列存储的不同之处在于，它针对的并不是传统的关系数据，而是针对嵌套结构的数据。Dremel 可以将一条条嵌套结构的记录转换成列存储形式，查询时根据查询条件读取需要的列，然后进行条件过滤，输出时再将列组装成嵌套结构的记录输出，记录的正向和反向转换都通过高效的状态机实现。另外，Dremel 的多层查询树则借鉴了分布式搜索引擎的设计，查询树的根节点负责接收查询，并将查询分发到下一层节点，底层节点负责具体的数据读取和查询执行，然后将结果返回上层节点。

Dremel 使用的数据就是我们熟悉的 Protocol Buffer 格式，但通常情况下我们将其作为序列化方法或者在 RPC 中传输等场景使用，较少用它来存放大量数据。对于没有接触过 Protocol Buffer 的读者，可以用 JSON 类比，二者的结构很相似，最大的不同是 Protocol Buffer 不支持 JSON 的 map（或者说是 dict、hashmap）。

一个 Protocol Buffer 的 Document.proto 文件示例如下：

```
message Document {
  required int64 DocId;
  optional group Links {
     repeated int64 Backward;
     repeated int64 Forward;
  }
```

```
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
    }
    optional string Url;
  }
}
```

需要注意的不是数据本身，而是数据的类型，或者说是数据的架构。但从中可以看出两个特点：类型是可以嵌套的；同一种类型的数据是可以重复（repeated）和可选（optional）的。

Dremel 可以使用一种 SQL-like 的语法查询嵌套数据。由于 Dremel 的数据是只读的，并且会密集地发起多次类似的请求，所以可以保留上次请求的信息，同时优化下次请求的 explain 过程。那又是如何 explain 的呢？我们来看看图 15-1。

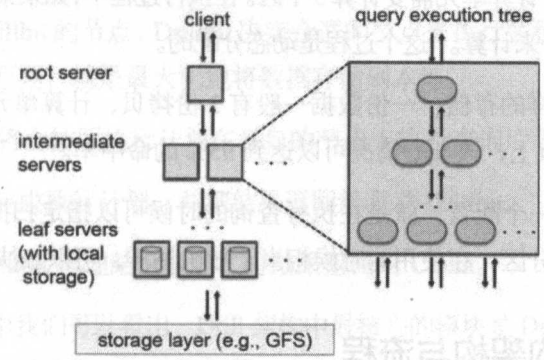


图 15-1　explain 过程

这是一个树状架构。Client 发起一个请求，根节点收到请求后，根据 MetaData 将其分解到枝叶，直到位于数据上面的叶子 Server。它们扫描处理数据，又不断汇总到根节点。

举个例子，对于如下请求：

```
SELECT A, COUNT(B) FROM T GROUP BY A
```

根节点收到请求后，会根据数据的分区请求，将请求变成可以拆分的叶子。原来的请求会变为：

```
SELECT A, SUM(c) FROM (R1 UNION ALL ... Rn) GROUP BY A
```

R1,…,Rn 是 T 的分区计算出来的结果集。越大的表有越多的分区，越多的分区可以越好地支持并发。

然后将请求切分，发送到每个分区的叶子 Server 上面去。对于每个 Server：

```
Ri = SELECT A, COUNT(B) AS c FROM Ti GROUP BY A
```

结果集一定会比原始数据小很多，处理起来也更快。根服务器可以很快地将数据汇总。具体的聚合方式可以使用现有的并行数据库技术。

Dremel 是一个多用户的系统。在切割分配任务的时候，还需要考虑用户优先级和负载均衡。对于大型系统，还需要考虑容错，如果一个叶子 Server 出现故障或变慢，不能让整个查询也受到明显影响。

通常情况下，每个计算节点执行多个任务。例如，有 3000 个叶子 Server，每个 Server 使用 8 个线程，可以有 24 000 个计算单元。如果一张表可以划分为 100 000 个区，就意味着大约每个计算单元需要计算 5 个区。在执行过程中，如果某个计算单元太忙，就会另外启动一个来计算。这个过程是动态分配的。

对于 GFS 这样的存储，一份数据一般有 3 份拷贝，计算单元很容易就能分配到数据所在的节点上，典型的情况可以达到 95% 的命中率。

Dremel 还有一个配置，就是在执行查询的时候可以指定扫描部分分区，比如可以扫描 30% 的分区，在使用的时候相当于随机抽样，可以加快查询。

15.3 Drill 的架构与流程

从 Drill 官方对其架构的介绍中得知，其具有适于实时的分析和快速的应用开发、适于半结构化 / 嵌套数据的分析、兼容现有的 SQL 环境和 Apache Hive 等特征。

Drill 的查询流程图如图 15-2 所示。

Drill 的查询流程大致如下：

（1）Drill 客户端发起一个查询。Drill 客户端是 JDBC、ODBC 命令行或者一个 REST API。Drill 中有一个核心模块，就是 Drillbit 服务。当 Drill 客户端发起查询后，被在 Cluster 中的 Drillbit 服务接收。需要注意的是，和其他大数据平台架构不同的是，Drill 不是 Master-Slave 模式。

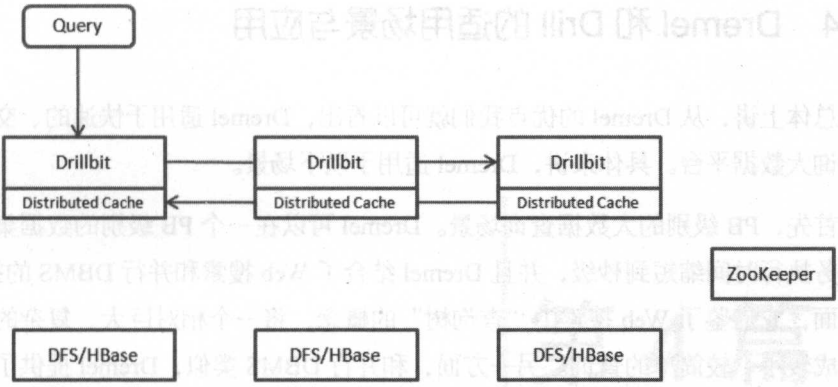


图 15-2　Drill 的查询流程图

（2）Drillbit 接收到查询之后，解析并优化这个查询，生成一个分布式查询计划，这个计划可以快速、有效地被执行。

（3）当 Drillbit 接收到查询后，就变成了驱动 Drillbit。随后，从 ZooKeeper 中获得可利用的 Drillbit 的节点，Drillbit 决定合适的节点来执行各种查询计划的片段。但是这里有一个原则，就是最大化地将数据存储到本地。

（4）Drillbit 将会按照执行计划在各自的节点上执行查询片段。

（5）各节点完成执行计划，并把结果返回给驱动 Drillbit。

（6）这个 Drillbit 梳理结束之后，将结果返回给 Client。

从这个流程中我们可以看出，Drill 架构中最核心的模块是 Drillbit 服务。该服务模块包括远程访问子模块、SQL 解析器、查询优化器、任务计划执行引擎、存储插件接口（DFS、HBase、Hive 等的接口）、分布式缓存模块等几部分，如图 15-3 所示。

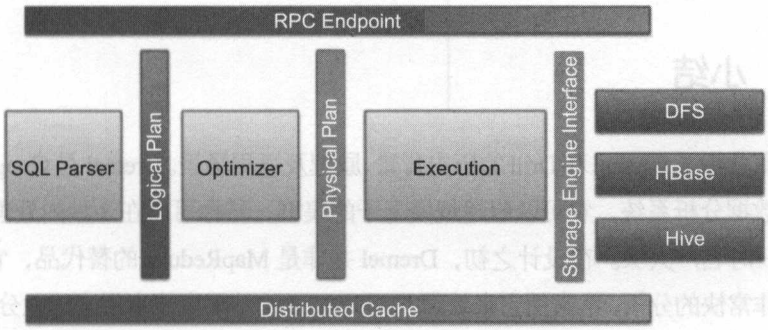


图 15-3　Drillbit 服务模块

15.4 Dremel 和 Drill 的适用场景与应用

总体上讲，从 Dremel 的优点我们就可以看出，Dremel 适用于快速的、交互式的查询大数据平台。具体来讲，Dremel 适用于以下场景。

首先，PB 级别的大数据查询场景。Dremel 可以在一个 PB 级别的数据集上面将任务执行时间缩短到秒级，并且 Dremel 结合了 Web 搜索和并行 DBMS 的技术。一方面，它借鉴了 Web 搜索中“查询树”的概念，将一个相对巨大、复杂的查询分割成较小、较简单的查询。另一方面，和并行 DBMS 类似，Dremel 提供了一个 SQL-like 的接口，就像 Hive 和 Pig 那样。

其次，当 MapReduce 用户需要更快的交互式查询的时候，Dremel 可以派上用场。和 MapReduce 一样，Dremel 也需要和数据运行在一起，将计算移动到数据上面，所以它需要 GFS 这样的文件系统作为存储层。在设计之初，Dremel 并非是 MapReduce 的替代品，它只是可以执行非常快的分析，常常用它来处理 MapReduce 的结果集或者用来建立分析原型。当然，这一点在 Dremel 的公开论文里面已经说得很明白，Dremel 不是用来替代 MapReduce 的，而是和其更好地结合。Hadoop 的 Hive、Pig 无法提供及时的查询，而 Dremel 的快速查询技术可以给 Hadoop 提供有力的补充。同时，Dremel 可以用来分析 MapReduce 的结果集，只需将 MapReduce 的 OutputFormat 修改为 Dremel 的格式，就可以几乎不引入额外开销而将数据导入 Dremel。一般使用 Dremel 来开发数据分析模型，使用 MapReduce 来执行数据分析模型。

就目前的应用而言，Google 提供了一个基于 Dremel 的网络服务 BigQuery，用户可以通过在线 API 使用这个平台。也就是说，只要你将数据上传至 Google，就可以利用 Google 的内部架构来执行查询。

15.5 小结

本章分析了 Dremel 和 Drill 的历史背景、原理及适用场景。Dremel 是 Google 的“交互式”数据分析系统，它可以组建规模上千的集群，号称可以在 3 秒内处理 1PB 的数据，一时名声大噪。在设计之初，Dremel 并非是 MapReduce 的替代品，它只是可以执行非常快的分析，常常用它来处理 MapReduce 的结果集或者用来建立分析原型。Drill 是基于 Dremel 实现的开源项目，目前已经成为 Apache 基金会的顶级项目。

第 4 篇

大数据下的日志分析系统

16 日志分析解决方案

16.1 百花齐放的日志处理技术

众所周知，大数据平台组件是很复杂的。笔者之前接触的一个大数据平台解决方案，仅平台组件就达 20 多个，这还没有加上物联网系统各组件。而这庞大的系统整合问题，对于运维来说是很头疼的。所以，在大数据时代下的运维问题日渐尖锐。

有人把运维比作医生给病人看病，那么日志则是病人对自己病情的陈述。所以只有在海量分布式日志系统中有效地提取关键信息，才能对症下药。如果能把这些日志集中管理，并提供全文检索功能，那么不仅可以提高诊断效率，而且可以提供实时系统监测、网络安全、事件管理和发现 Bug 等功能。

说到日志分析工具，目前此类工具已经有几十种，甚至上百种，面对不同的场合应用不同的技术。有商业性质的软件，也有免费的，下面我们对业界比较热门的日志处理软件进行简单介绍。

1. Splunk

在商业形式的软件中，最具代表性的是 Splunk。Splunk 是一款顶级的日志分析软件，提供收集、处理、展现一条龙服务。如果你经常用 grep、awk、sed、

sort、uniq、tail、head 来分析日志，那么你需要 Splunk。Splunk 能处理常规的日志格式，比如 Apache、Squid、系统日志、mail.log 等。首先对所有日志进行索引，然后可以交叉查询，支持复杂的查询语句，最后通过直观的方式表现出来。日志可以通过文件方式传到 Splunk 服务器上，也可以通过网络实时传输过去，或者是分布式的日志收集。总之，它支持多种日志收集方法。据说其价格不菲，如果是企业用户，则可以考虑。

2. AWStats

AWStats 是在 Sourceforge 上发展很快的一个基于 Perl 的 Web 日志分析工具。AWStats 界面友好，可以根据浏览器直接调用相应的语言界面，有简体中文版。AWStats 效率比较高，并且配置 / 定制方便。最主要的是，AWStats 是基于 Perl 的，很好地解决了跨平台问题，系统本身可以运行在 GNU/Linux 或 Windows 上。AWStats 能分析来自各大服务器工具的日志文件，如 Apache 日志档案，WebStar，IIS（W3C 的日志格式）及其他网站，Proxy（代理服务器）、WAP、流量服务器、邮件服务器和 FTP 服务器的日志文件。

3. Graphite

Graphite 是一个用于采集网站实时信息并进行统计的开源项目，可用于采集多种网站服务运行状态信息。Graphite 服务平均每分钟有 4800 次更新操作。众所周知，检测网站对于系统管理员和维护人员来说是十分必要的。Graphite 的简单文本协议和绘图功能可以方便地以即插即用的方式用于任何需要监控的系统上。

4. LogAnalyzer

LogAnalyzer 是一个 Web 前端工具，提供简单易用的日志浏览、搜索和基本分析及图表显示。数据可以从数据库或一般的 syslog 文本文件中获取，所以 LogAnalyzer 不需要改变现有的记录架构。基于当前的日志数据，它可以处理 syslog 日志消息、Windows 事件日志记录，支持故障排除。

5. Rsyslog

Rsyslog 可以理解为 syslogd 工具的多线程增强版。在流行的 Linux 发布的版本中，都携带了这个工具。该工具具有各种各样的新功能，典型的有：（1）直接将日志写入数据库；（2）日志队列（内存队列和磁盘队列）；（3）灵活的模板机制，

可以得到多种输出格式；（4）插件式结构，多种多样的输入、输出模块。

6. Open Web Analytics

Open Web Analytics 是一个开源的网站流量统计系统。它基于 PHP/Open Flash Chart/Ajax 技术开发，既可以单独使用，也可以与 WordPress、Gallery&MediaWiki 集成使用。它支持多个网站，集成了 Google Maps、RSS/Atom 订阅跟踪等功能。

7. Logwatch

Logwatch，顾名思义，是一款专门监测 Linux log 文件（日志文件）的软件。安装以后只要稍微配置一下，就能每天将主机的 log 分析文件发送至指定的邮箱。为什么要这么做？因为这样能减少手动查看日志的烦琐工作，为管理员节省时间和精力，并且每天（甚至更频繁）都能准确地接收到日志，从而对服务器的安全情况有所掌握。

8. GlTail.rb

GlTail.rb 是一款带有浓郁的 Geek 风格的可视化日志分析工具，它采用 Ruby 技术构建，并利用 OpenGL 图形技术进行渲染，呈现可视化的日志分析数据，以气泡数量来反映网站的访问情况。

以上这些软件，读者如果有需要，可以详细研究如何应用。接下来，我们请出本章的重点，也就是笔者要介绍的，集日志收集、处理、分析、存储、展现等功能于一体的 ELK。实际上它不是一个工具，而是一个解决方案，是几个工具的结合。

16.2 日志处理方案 ELK

16.2.1 ELK 的三大金刚

ELK 是 ElasticSearch、Logstash、Kibana 的简称，各取这三个工具的首字母，便构成了 ELK。这三个工具是核心套件，但并非全部。这三个软件均是开源的。ELK Stack 在最近两年迅速崛起，成为机器数据分析或者实时日志处理领域开源界的第一选择。接下来分别看看这三个工具到底是什么，以及它们的主要职责。

1. Logstash

Logstash 是一个接收、处理、转发日志的工具，支持系统日志、WebServer 日志、错误日志、应用日志，总之，包括所有可以抛出来的日志类型。Logstash 在其过程中担任搬运工的角色，它为数据存储、报表查询和日志解析创建了一个功能强大的管道链。Logstash 提供了多种多样的 input、filters、codecs 和 output 组件，让使用者可以轻松实现强大的功能。它可以从许多来源接收日志，这些来源包括 syslog、消息传递（例如 RabbitMQ）和 jmx。它能够以多种方式输出数据，包括电子邮件、WebSockets 和 ElasticSearch。Logstash 的主要职责是采集日志、解析日志，并将解析后的日志写入 ElasticSearch 中。

2. ElasticSearch

ElasticSearch 是一个基于 Lucene 的搜索服务器，具有实时全文搜索和分析引擎，提供搜集、分析、存储数据三大功能。它是基于 RESTful Web 接口，具有分布式多用户检索能力的全文搜索引擎。ElasticSearch 是用 Java 语言开发的，并作为 Apache 许可条款下的开放源码发布，是第二流行的企业搜索引擎。它设计用于云计算中，能够达到实时搜索，稳定、可靠、快速，安装、使用方便。据报道，ElasticSearch 已经应用到了数十亿级别的大数据处理环境中。2013 年年初，GitHub 放弃 Solr，采取 ElasticSearch 来做 PB 级的搜索，包括 13 亿个文件和 1300 亿行代码。

3. Kibana

Kibana 是一个基于 Web 的图形界面，用于搜索、分析和可视化存储在 ElasticSearch 指标中的日志数据。它利用 ElasticSearch 的 REST 接口来检索数据，不仅允许用户创建自己数据的定制仪表板视图，还允许用户以特殊的方式查询和过滤数据。

实际上，在整个 ELK 体系中，面对不同的搭建场景，可能会涉及其他的工具或者软件，可能用到的软件还有 Redis（一个 Key-Value 的数据库）、Logstash-forwarder（协助 Logstash 收集 log）、Beats（Packetbeat、Topbeat 和 Filebeat）。我们稍后还会对这几种工具展开详细的讨论。

和传统的日志处理方案相比，ELK Stack 具有如下几个优点：

- 处理方式灵活。ElasticSearch 是实时全文索引，不需要像 Storm 那样预先编程才能使用。
- 配置简单，易上手。ElasticSearch 全部采用 JSON 接口，Logstash 由 Ruby DSL 设计，二者都是目前业界通用的配置语法设计。
- 检索性能高效。虽然每次查询都是实时计算，但是优秀的设计和实现基本可以达到全天数据查询的秒级响应。
- 集群线性扩展。不管是 ElasticSearch 集群还是 Logstash 集群，都是可以线性扩展的。
- 前端操作炫丽。在 Kibana 界面上，只需单击鼠标，就可以完成搜索、聚合功能，生成炫丽的仪表板。

16.2.2 ELK 的架构

现在我们清楚了各个工具的主要职责，接下来看看它们在整个 ELK 系统中的位置。经典的 ELK 系统的架构如图 16-1 所示。

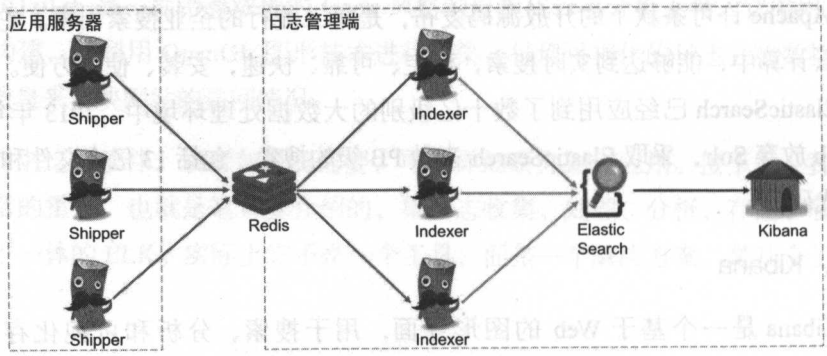


图 16-1　经典的 ELK 系统的架构

从 ELK 系统的架构图中可以看出，其主要包括以下几个部分。

1. Shipper

Shipper 是货主、发货人、托运人的意思。这里的货物是“日志”，可以通俗地理解为：存储日志，并且把日志“发出去”。

Shipper 主要负责日志的采集，采集的形式多种多样，可以从文件中采集，也可以从程序接口中采集。图 16-1 中用 Logstash 的图标来作为 Shipper，也就是说

用 Logstash 来采集数据。之前我们提到，Logstash 既有采集日志的功能，又有解析处理日志的功能。但在实际应用中，很少场合把 Logstash 部署到客户端去采集数据，因为 Logstash 在运行过程中，内存和 CPU 消耗都比较大。客户端要集中最大的能量来执行任务，如果因为管理或者协助管理的进程占用了太多的资源，那将是本末倒置、得不偿失的。

那么，在实际的 ELK 环境中，都有哪些工具可用来充当 ELK 呢？可以是 Beats（Topbeat、Filebeat、Packetbeat，以及自己开发的 beat）、Logstash-forwarder、Ssylog（或者 Rsyslog）等，也可以是自己开发的工具。总之，只要发给下一站 Broker（或者直接给 Indexer）就可以了。

2. Broker

Broker 是“经纪人”、“中间人”的意思。在 ELK 的架构中，这个环节的确充当了中间人的作用。当然，这个中间人可有可无，如果没有中间人，那么 Shipper 直接将数据发送给下一站 Indexer；如果有，通常也和 Indexer 部署在一个实体上。那么为什么在实际的组网中可以不专门设置 Broker 呢？因为如果服务器性能较好，则 Logstash 和 ElasticSearch 完全可以实时解析并存储日志。

Broker 负责对数据进行缓存，在数据被解析之前先寄存在这里。一般情况下使用 Redis 或者 Kafka 等 Key-Value 型的数据库。

3. Indexer

Indexer 主要用来索引、解析、分析接收到的数据。一般情况下，由 Logstash 来担任 Indexer 的职责。解析之后，按照规定的格式发送给下一站 ElasticSearch。当然，Logstash 的 output 不一定是 ElasticSearch，可以是一个普通的文件，也可以是存储系统，甚至是终端。

4. Storage&Search

这里会使用 ElasticSearch。ElasticSearch 负责存储日志并提供检索功能。

5. Web Interface

一般用 Kibana 来展现。可以以图的形式展示，也可以以文字的形式展示，多用于 Dashboard 的展示。总之，可以理解为我们平时提到的 GUI。

16.2.3 ELK 的组网形式

在经典的 ELK 系统框架之下，又衍生出了几种 ELK 的组网。

1. Logstash 采集数据模式

在这种模式下，Logstash 充当了 Shipper 和 Indexer，采集和解析工作全部由 Logstash 来做。此架构首先由 Logstash 分布于各个节点上搜集相关日志、数据，并经过分析、过滤后发送给远端服务器上的 ElasticSearch 进行存储。ElasticSearch 将数据以分片的形式压缩存储并提供多种 API 供用户查询、操作。用户也可以更直观地通过配置 Kibana Web Portal 方便地对日志进行查询，并根据数据生成报表。Logstash 采集数据模式如图 16-2 所示。

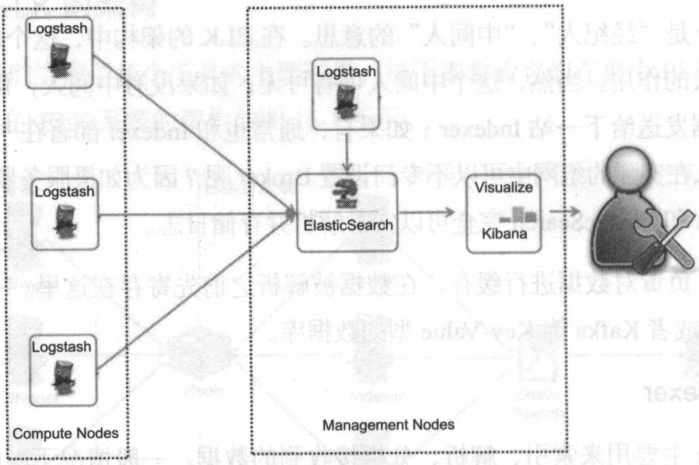


图 16-2　Logstash 采集数据模式

这种模式的优点在于组网简单、配置容易、上手快。但是其缺点也显而易见，Logstash 耗费资源较大，运行占用 CPU 和内存高，如果把 Logstash 放到客户端，则可能会影响集群的性能。另外，没有消息队列缓存，存在数据丢失隐患。建议供学习者和小规模集群使用。

2. 消息队列模式

这种架构引入了消息队列机制，位于各个节点上的 Logstash Agent 先将数据 / 日志传递给 Kafka（或者 Redis），并将队列中的消息或数据间接传递给 Logstash；Logstash 过滤、分析后将数据传递给 ElasticSearch 存储，最后由 Kibana 将日志和

数据呈现给用户。因为引入了 Kafka（或者 Redis），所以即使远端 Logstash Server 因故障停止运行，数据也会被存储下来，从而避免了数据丢失。消息列队模式如图 16-3 所示。

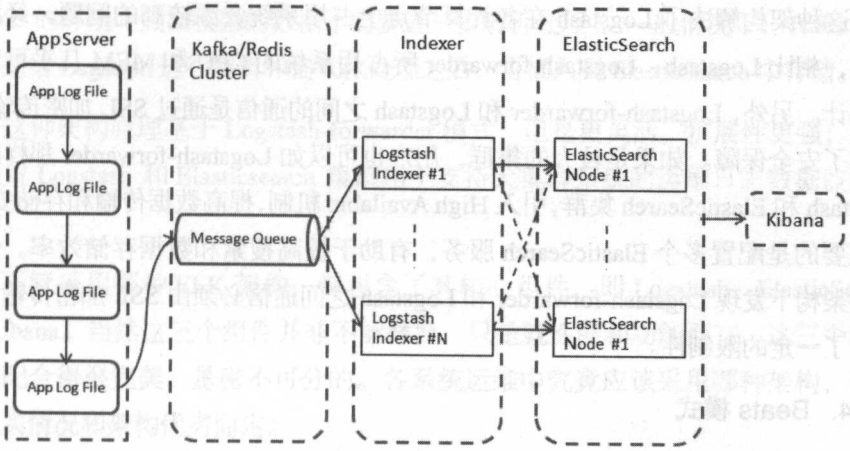


图 16-3 消息队列模式

这种架构适合于较大集群的解决方案，但由于 Logstash 中心节点和 ElasticSearch 的负荷会比较重，可将它们配置为集群模式，以分担负荷。这种架构的优点在于引入了消息队列机制，均衡了网络传输，从而降低了网络闭塞尤其是丢失数据的可能性，但依然存在 Logstash 占用系统资源过多的问题。

3. Logstash-forwarder 模式

Logstash-forwarder 是一个轻量级的工具，用来辅助 Logstash 收集数据，如图 16-4 所示。在这种模式中，Shipper 端由 Logstash-forwarder 来充当，Logstash-forwarder 和 Logstash 之间采用 Lumberjack 协议沟通；相应地，在 Logstash 端也提供了 Lumberjack 的插件。首先，Logstash-forwarder 将日志数据搜集并统一发送给主节点上的 Logstash，Logstash 分析、过滤日志数据后发送至 ElasticSearch 存储，并由 Kibana 最终将数据呈现给用户。

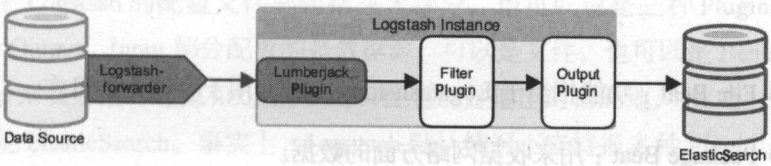


图 16-4 Logstash-forwarder 模式

值得一提的是，因为采用的是 Lumberjack 协议，所以 Logstash-forwarder 和 Logstash 之间是密文传输，在搭建环境之前，需要事先准备好 SSL 协议所需要的证书。并且这种模式逐渐被 Beats 模式所替代，相应的代码也不再维护。

这种架构解决了 Logstash 在各计算节点上占用系统资源较高的问题。经测试得出，相比 Logstash，Logstash-forwarder 所占用系统的 CPU 和 MEM 几乎可以忽略不计。另外，Logstash-forwarder 和 Logstash 之间的通信是通过 SSL 加密传输的，起到了安全保障。如果是较大的集群，用户也可以如 Logstash-forwarder 那样配置 Logstash 和 ElasticSearch 集群，引入 High Available 机制，提高数据传输和存储安全。更重要的是配置多个 ElasticSearch 服务，有助于提高搜索和数据存储效率。但在此种架构下发现 Logstash-forwarder 和 Logstash 之间通信必须由 SSL 加密传输，这便有了一定的限制性。

4. Beats 模式

经测试，Beats 满负荷状态所耗系统资源和 Logstash-forwarder 相当，但其扩展性和灵活性有很大提高。Beats Platform 目前包含 Package Beat、Top Beat 和 File Beat 三个产品，均为 Apache 2.0 License。同时用户可以根据需要进行二次开发。Beats 模式如图 16-5 所示。

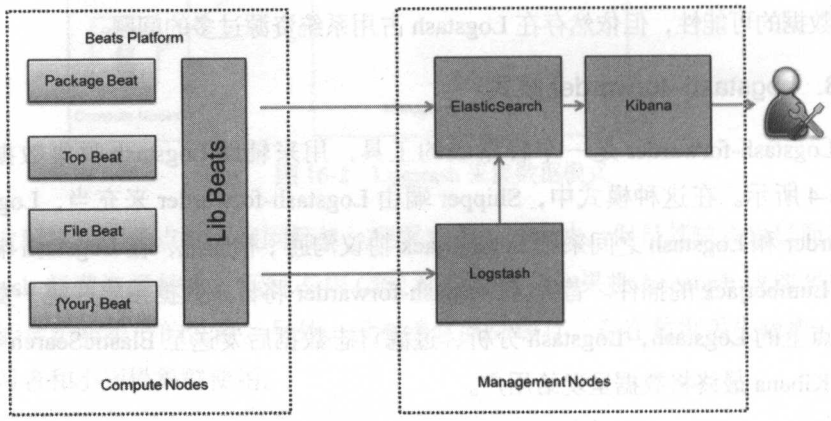


图 16-5　Beats 模式

（1）File Beat：功能相当于 Logstash-forwarder，用于收集文件日志。

（2）Package Beat：用来收据网络方面的数据。

（3）Top Beat：用来收集机器性能方面的数据，比如CPU、内存、磁盘等相关信息。

从图16-5中可以看出，Beats可以直接将数据发送给ElasticSearch，省略了Indexer的环境（假如收集的数据不需要进一步解析处理）。一般情况下，Teabs将数据发送给Logstash进行解析和格式化处理之后，再插入到ElasticSearch中存储。

这种架构原理基于Logstash-forwarder模式，但是更灵活，扩展性更强，同时可配置Logstash 和 Elasticsearch 集群用于支持大集群系统的运维日志数据监控和查询。

不管采用哪种ELK架构，都包含了其核心组件，即Logstash、ElasticSearch和Kibana。当然这三个组件并非不能替换，只是就性能和功能而言，这三个组件已经配合得很完美，是密不可分的。各系统运维中究竟应该采用哪种架构，可根据现实情况和架构优劣而定。

16.3 Logstash 日志收集解析

Logstash作为ELK体系中的一员骨干，担负着收集和分析日志的重任。从数据源采集到数据后，经过分析，发送给ElasticSearch，如图16-6所示。

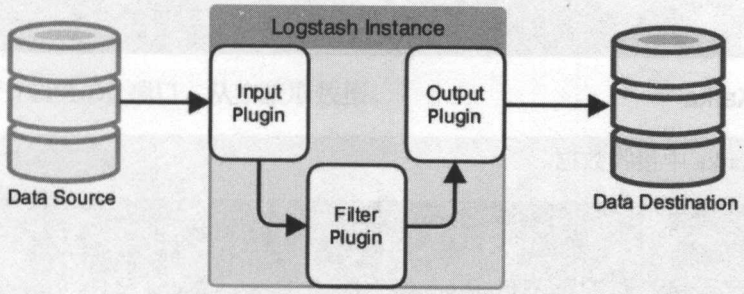


图16-6 Logstash结构

每个Logstash的配置文件都包括三大部分，也可以说是三种Plugin：Input、Filter和Output。Input部分配置的是数据源，可以是文件，也可以是Beats等输出源。Filter对日志进行过滤和处理。Output配置的是输出的目的地，可以是一个文件，也可以是ElasticSearch。事实上，Logstash默认情况下支持很多种Plugin。下文我们将讨论很常见的Plugin。

16.3.1 Input Plugins 及应用示例

1. Beats

这应该是最常用的 Input Plugin，一般情况下，在计算节点上都用 Beats 来做客户端。以下是一个最简单的 Beats Input 配置：

```
input {
  beats {
    port => "5043"
    ssl_certificate => "/opt/key/lumberjack.crt"
    ssl_key => "/opt/key/lumberjack.key"
  }
}
```

2. File

如果用 Logstash 来直接收集日志，那么这个 Input Plugin 就派上用场了。可以用这个 File 收集本地的日志文件。

```
input {
 file {
  type => "linux-syslog"
  path => [ "/var/log/messages" ]
  start_position => "beginning"
 }
}
```

3. Kafka

从 Kafka 中接收数据。

```
input {
  kafka {
    zk_connect => "localhost:2181"
    group_id => "logstash"
    topic_id => "test"
    reset_beginning => false # boolean (optional), default: false
    consumer_threads => 5  # number (optional), default: 1
    decorate_events => true # boolean (optional), default: false
    }
  }
```

4. Redis

从 Redis 中接收数据。

```
input {
       redis{
       host => "localhost"
       data_type => "pattern_channel"
        key => "logstash:*"
       tags => "redis_filter_proxy"
    }
}
```

5. Stdin

从输出流中接收数据，主要用于捕捉系统的错误流信息。这应该是 Logstash 里最简单和最基础的插件。

```
input {
  stdin {
     add_field => {"key" => "value"}
     codec => "plain"
     tags => ["add"]
     type => "std"
  }
}
```

6. TCP/UDP

监听 TCP/UDP 端口，从中获取数据。

```
input {
  tcp {
      port => 8888
      mode => "server"
      ssl_enable => false
  }
}
```

7. Generator

实际运行的时候这个插件是派不上用场的，但是这个插件依然非常重要。因为每一个使用 ELK Stack 的运维人员都应该清楚一个道理：数据是支持操作的唯一真理（否则你也用不着 ELK）。所以在上线之前，你一定会在自己的实际环境中

测试 Logstash 和 ElasticSearch 的性能状况。这时候，这个用来生成测试数据的插件就有用了。

```
input {
  generator {
      count => 10000000
      message=>
'{"key1":"value1","key2":[1,2],"key3":{"subkey1":"subvalue1"}}'
      codec => json
  }
}
```

16.3.2 Filter Plugins 及应用示例

1. Date

该插件可以用来转换日志记录中的时间字符串，变成 LogStash::Timestamp 对象，然后转存到 @timestamp 字段里。

```
date {
  match => ["logdate", "dd/MMM/yyyy:HH:mm:ss Z"]
  timezone => "UTC"
}
```

2. Drop

如果不满足某种过滤条件，则可以将此数据丢弃。

```
if "_grokparsefailure" in [tags] { drop { } }
```

3. Grok

Grok 是 Logstash 最重要的插件。你可以在 Grok 里预定义命名正则表达式，稍后（Grok 参数或者其他正则表达式里）再引用它。

```
filter {
  grok {
     match => {
        "message" => "\s+(?<request_time>\d+(?:\.\d+)?)\s+"
     }
  }
}
```

4. Mutate

Mutate 是 Logstash 另一个重要的插件，它提供了丰富的基础类型数据处理能力，包括类型转换、字符串处理和字段处理等。

```
filter {
  mutate {
    convert => ["request_time", "float"]
  }
}
```

5. Multiline

合并多行数据，将多行数据归结为一条日志。

```
filter {
  multiline {
    pattern => "^[^\[]"
    what => "previous"
  }
}
```

6. Ruby

如果你懂一点 Ruby 语法，那么 Ruby 插件将会是一个非常有用的工具。比如你需要稍微修改一下 LogStash::Event 对象，但是又不打算为此写一个完整的插件，Ruby 插件绝对是最佳选择。

```
filter {
  ruby {
      init => "@kname =
['client','servername','url','status','time','size','upstream','upstre
amstatus','upstreamtime','referer','xff','useragent']"
      code => "event.append(Hash[@kname.zip(event['message'].split('|'))])"
  }
}
```

16.3.3 Output Plugins 及应用示例

1. ElasticSearch

将解析后的数据保存到 ElasticSearch 中是最常见的一种输出。Logstash 早期有三个不同的 ElasticSearch 插件，到 1.4.0 版本的时候，开发者彻底重写了

LogStash::Outputs::Elasticsearch 插件。从此，我们只需要一个插件，就能任意切换使用 ElasticSearch 集群支持的各种不同协议。现在，新插件支持三种协议：Node、HTTP 和 Transport。在最新版本中，默认支持的协议是 HTTP。

```
output {
    elasticsearch {
        host => "192.168.0.2"
        protocol => "http"
        index => "logstash-%{type}-%{+YYYY.MM.dd}"
        index_type => "%{type}"
        workers => 5
        template_overwrite => true
    }
}
```

2. File

将解析后的数据保存到文件中。通过日志收集系统将分散在数百台服务器上的数据集中存储在某中心服务器上，这是运维最原始的需求。

```
output {
    file {
      path => "/path/to/%{+yyyy/MM/dd/HH}/%{host}.log.gz"
      message_format => "%{message}"
      gzip => true
    }
}
```

3. Stdout

Stdout 插件也是最基础和最简单的输出插件。它将过滤后的结果直接打印出来，主要用于定位问题。

```
stdout { codec => rubydebug { } }
```

注意，如果需要打印 MetaData，则配置如下：

```
stdout { codec => rubydebug { metadata => true } }
```

16.4 ElasticSearch 存储与搜索

近年来 ElasticSearch 发展迅猛，已经超越了其最初的纯搜索引擎的角色，增

加了数据聚合分析（Aggregation）和可视化的特性。如果你有数百万篇的文档需要通过关键词进行定位，则 ElasticSearch 肯定是最佳选择。当然，如果你的文档是 JSON 格式的，你也可以把 ElasticSearch 当作一种“NoSQL 数据库”，运用 ElasticSearch 数据聚合分析（Aggregation）的特性，针对数据进行多维度的分析。

百度目前广泛使用 ElasticSearch 作为文本数据分析，采集百度所有服务器上的各类指标数据及用户自定义数据，通过对各种数据进行多维分析展示，辅助定位分析实例异常或业务层面异常。目前覆盖百度内部 20 多条业务线（包括 casio、云分析、网盟、预测、文库、直达号、钱包、风控等），单集群最大 100 台机器，200 个 ES 节点，每天导入 30TB+ 数据。

16.4.1 ElasticSearch 的主要概念

要理解 ElasticSearch，先来看看一些重要的概念和组织成员。

1. Cluster 和 Node

ElasticSearch 可以以单点或者集群方式运行，以一个整体对外提供 Search 服务的所有节点组成 Cluster，组成这个 Cluster 的各个节点叫作 Node。

2. Index

这是 ElasticSearch 存储数据的地方，类似于关系数据库的 Database。

3. Shard

索引分片，这是 ElasticSearch 提供分布式搜索的基础，其含义为将一个完整的 Index 分成若干部分存储在相同或不同的节点上，这些组成 Index 的部分就叫作 Shards。

4. Replicas

索引副本，ElasticSearch 可以设置多个索引的副本。副本的作用一是提高系统的容错性，当某个节点某个分片损坏或丢失时可以从副本中恢复；二是提高 ElasticSearch 的查询效率，ElasticSearch 会自动对搜索请求进行负载均衡。

5. Recovery

代表数据恢复或称数据重新分布，ElasticSearch 在有节点加入或退出时会根据

机器的负载对索引分片进行重新分配，挂掉的节点重新启动时也会进行数据恢复。

6. Gateway

ElasticSearch 索引快照的存储方式。ElasticSearch 默认先把索引存放到内存中，当内存不足时再持久化到本地硬盘。Gateway 对索引快照进行存储，当这个 ElasticSearch 集群关闭再重新启动时就会从 Gateway 中读取索引备份数据。

7. Discovery.zen

代表 ElasticSearch 的自动发现节点机制。ElasticSearch 是一个基于 P2P 的系统，它先通过广播寻找存在的节点，再通过多播协议来进行节点之间的通信，同时支持点对点的交互。

8. Transport

代表 ElasticSearch 内部节点或集群与客户端的交互方式。默认内部使用 TCP 协议进行交互，同时支持 HTTP 协议（JSON 格式）。

16.4.2 ElasticSearch Rest API

Restful 的接口很简单，一个 URL 表示一个特定的资源，如 /blog/article/1 就表示一个 Index 为 blog、Type 为 aritcle、ID 为 1 的 Document。ElasticSearch 支持通过 HTTP 请求响应服务，因此通过 curl 命令可以发送 HTTP 请求，并得到 JSON 返回内容。

我们使用 HTTP 标准 Method 来操作这些资源，POST 表示新增，PUT 表示更新，GET 表示获取，DELETE 表示删除，HEAD 表示判断是否存在。常用的 Rest 请求如下。

检查 ElasticSearch 集群状态：

```
curl localhost:9200/_cat/health?v
```

检查 ElasticSearch 节点状态：

```
curl localhost:9200/_cat/nodes?v
```

查询所有索引：

```
curl localhost:9200/_cat/indices?v
```

创建索引：

```
curl -XPUT localhost:9200/索引名字/类型/id -d {"name":"xingoo"}
```

删除索引：

```
curl -XDELETE localhost:9200/索引名字
```

查询索引：

```
curl -XGET localhost:9200/索引名字/类型名字/id
```

16.5 Kibana 展示

Kibana 是为 ElasticSearch 设计的开源分析和可视化平台，可以用来搜索、查看存储在 ElasticSearch 索引中的数据并与之交互。你可以很容易地实现高级的数据分析和可视化，并以图表的形式展现出来。Kibana 让海量数据变得更容易理解。简单的基于浏览器的界面可以让你快速创建并分享动态的仪表板，用以实时修改 ElasticSearch 请求。

安装 Kibana 非常简单。你可以在几分钟内安装好 Kibana 然后开始探索你的 ElasticSearch 索引——不需要写代码，不需要额外的架构。在开始使用 Kibana 之前，你需要告诉它你打算探索哪个 ElasticSearch 索引。第一次访问 Kibana 的时候，你会被要求定义一个 Index Pattern 用来匹配一个或者多个索引名。

Kibana 4 采用 angular.js + node.js 框架编写。其中 node.js 主要提供两部分功能，即给 ElasticSearch 做搜索请求转发代理，以及 Auth、SSL、Setting 等操作的服务器后端。

从浏览器访问 Kibana 界面，也就是说访问 localhost:5601 或者 http://KIBANA-SERVICE-IP:5601。访问 GUI 界面之后，首先需要创建 Index Pattern，如图 16-7 所示。

（1）指定一个可以匹配一个或者多个 ElasticSearch 索引的 Index Pattern。在默认情况下，Kibana 认为你要访问的是通过 Logstash 导入 ElasticSearch 的数据。这时候你可以用默认的 logstash-* 作为你的 Index Pattern。通配符（*）匹配索引名中的多个字符。如果你的 ElasticSearch 索引有其他命名约定，则输入合适的 Pattern。Pattern 一开始是最简单的单个索引的名字。

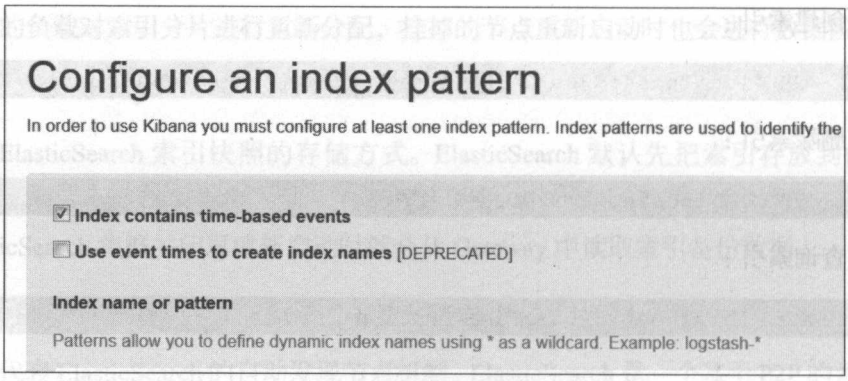


图 16-7　Kibana 展示

（2）选择一个包含了时间戳的索引字段，可以用来进行基于时间的处理。Kibana 会读取索引的映射，然后列出所有包含了时间戳的字段（译者注：实际是字段类型为 date 的字段，而不是“看起来像时间戳”的字段）。如果你的索引没有基于时间的数据，则关闭 Index contains time-based events 参数。

（3）如果一个新索引是定期生成的，而且索引名中带有时间戳，则先选择 Use event times to create index names 选项，然后再选择 Index pattern interval，这样可以提高搜索性能。Kibana 会搜索你指定时间范围内的索引。在你用 Logstash 输出数据给 ElasticSearch 的情况下尤其有效。

（4）单击 Create 添加 Index Pattern。第一个被添加的 Pattern 会自动被设置为默认值。如果你有多个 Index Pattern，则可以在 Settings → Indices 里设置具体哪个是默认值。

在默认情况下，Kibana 会连接运行在本地服务器上 的 ElasticSearch。要连接其他 ElasticSearch 实例，需修改 kibana.yml 里的 Elasticsearch URL，然后重启 Kibana。Kibana 连接上 ElasticSearch 数据后，会显示匹配上的索引里的字段名的只读列表。

我们可以用 Kibana 画各种图，比如区块图（Area Charts）、线性图（Lines Charts）、数据表格（Data Table）、Markdown 挂件（Markdown Widget）、饼图（Pie Charts）、竖条图（Vertical Bar Charts）、瓦片地图（Tile Maps）等。

16.6 小结

通常日志被分散存储在不同的设备上，因而集中化的日志管理显得很重要。ELK 无疑是目前最流行的集日志收集、处理、存储、Web 展现于一体的技术，而且 ELK 是开源的。本章展现了 ELK 的各种组网形式，详细阐述了三大组件（Logstash、ElasticSearch 和 Kibana）的应用，其中最为核心的是 ElasticSearch。

17 ELK 集群部署与应用

17.1 ELK 集群部署与优化

17.1.1 ELK HA 集群部署

我们以 ELK 监控 Spark 为例，来看看 ELK 集群监控系统的部署和优化过程。为了便于分析，我们将 Spark 集群分为管理节点（Master Node）和计算节点（Computer Node）。在图 17-1 中，我们考虑了 HA（High Available，高可用性）的情况。

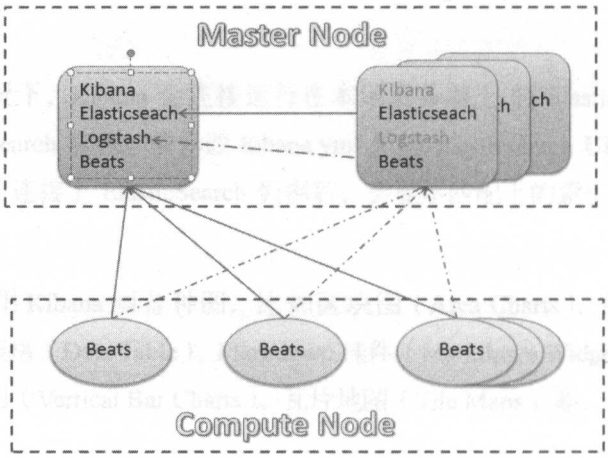


图 17-1　ELK HA 结构

管理节点（Master Node）可能有多个节点，分别安装 Beat、Logstash、ElasticSearch 和 Kibana（至于怎么安装，官方提供了 TAR 包和 RPM 包，非常简单，可以参考官方文档或者网络上的各种攻略）。计算节点上只需安装 Beats 来收集日志即可。

在大数据处理安装中，HA 是很重要的一个环节。就 ElasticSearch 而言，其本身就具备 HA 能力。大体上讲，HA 可以分为两种：一种是主备模式（Active-standby）模式；另一种是负载均衡（Load Balance）模式。二者的区别在于，Active-standby 模式是主节点出现故障，备用节点才启用，继续主节点的进程运行；而 Load Balance 模式是，大家一起上，谁空闲了或者谁的资源多了，就把活分给谁干。还有一种就是把这二者结合起来，达到双璧合一的效果。作为 ELK 的集群监控系统，最好的方式是采用第三种方法。其中 ElasticSearch 最好采用 Load Balance 模式，在 Master Node 上进行负载均衡。Logstash 当然也可以采用负载均衡的方式，但是由于前文中讲过，Logstash 运行后占用资源（CPU 利用率和内存）较多，所以笔者建议，如果 Master Node 比较繁忙，不建议在所有 Master Node 上启动 Logstash。Kibana 当然无须全部启动了，采用 Active-standby 模式即可。Beats 在所有节点上都启动，因为要收集所有节点上的日志。

17.1.2 ElasticSearch 优化

在上面部署的 ELK 集群中，最为复杂的是 ElasticSearch，最具有优化空间的也是 ElasticSearch，其他三个组件（Logstash-kibana 和 Shipper）只是配置问题。所以一般情况下谈到 ELK 的优化，就需要在 ElasticSearch 上下功夫。可以从以下几个方面来优化 ElasticSearch。

1. 内存优化

如果你的 ElasticSearch 运行在专用服务器上，根据工程上的经验值，可分配一半内存给 ElasticSearch，另一半用于系统缓存。

可以通过修改 ES_HEAP_SIZE 环境变量来改变这个设定。在启动 ElasticSearch 之前将这个变量设置为预期值。另一个选择是修改 ElasticSearch 的 ES_JAVA_OPTS 变量，这个变量是在启动脚本（elasticsearch.in.sh 或 elasticsearch.bat）里传递的。你必须找到 -Xms 和 -Xmx 参数，它们是分配给进程的最小和最大内存。建

议设置成相同大小。其实设置 ES_HEAP_SIZE 就是设定可供 ElasticSearch 使用的内存大小。设置最大值和最小值的方式相同，命令如下：

```
$bin/elasticsearch -f -Xmx4g -Xms4g -Des.index.storage.type=memory
```

另外，ElasticSearch 默认使用的 GC 是 CMS GC，如果内存大小超过 6GB，则 CMS 很容易出现 stop-the-world，建议使用 G1 GC。

2. 设置合理的 Shards 和 Replicas

如果有多台机器，则可以采用每台机器设置 *n* 个 Shards 的方式。根据业务情况，可以考虑取消 Replicas。

```
curl -XPUT 'http://10.1.*.*:9200/dw-search/' -d '{
    "settings" : {
        "number_of_shards" : 5,
        "number_of_replicas" : 0
    }
}'
```

这里设置 5 个 Shards，复制为 0。如果需要 Replicas，则可以完成 Index 后再修改为 Replicas≥1，并减小 Shards 刷新间隔，操作如下：

```
curl -XPUT 'http://10.1.*.*:9200/dw-search/_settings' -d '{
"index" : {
        "refresh_interval" :"-1"
    }
}'
```

完成 Bulk 插入后再修改为初始值。

```
curl -XPUT 'http://10.1.*.*:9200/dw-search/_settings' -d '{
    "index" : {
        "refresh_interval" : "1s"
    }
}'
```

3. 去掉 mapping 中的 _all 域

Index 中默认会有 _all 域，会给查询带来方便，但是会增加索引时间和索引尺寸。去掉 mapping 中的 _all 域采用如下方法：

```
"_all" : {"enabled" : false}
```

4. 合理配置主节点和数据节点

配置文件：conf/elasticsearch.yaml。

```
node.master: true
node.data: true
```

（1）当 master 为 false，而 data 为 true 时，会对该节点产生严重负荷。

（2）当 master 为 true，而 data 为 false 时，该节点作为一个协调者。

（3）当 master 为 false，data 也为 false 时，该节点就变成了一个负载均衡器。

5. 多线程

多线程程序插入，可以根据服务器情况开启多个线程 Index，速度可以提高 n 倍，$n \geq 2$。

线程池默认使用固定值，使用 cached 有可能控制不好。主要是因为比较大的分片 relocation 时，会导致分片自动下线，集群可能处于危险状态。在集群高压时，如果使用 cached，分片也可能自动下线。

6. 设置 source 为压缩模式或者 disable

如果将 compress 设置为 true，则将大大减小 Index 占用的空间。或者设置 disable 参数后，_source 域将被忽略，不再存储。

17.2 如何开发自己的插件

在运行 Logstash 的时候，可以通过 --pluginpath 参数来加载自己写的插件。那么，插件又该怎么写呢？

1. 插件格式

一个标准的 Logstash 输入插件格式如下：

```
require 'logstash/namespace'
require 'logstash/inputs/base'
class LogStash::Inputs::MyPlugin < LogStash::Inputs::Base
  config_name 'myplugin'
  milestone 1
```

```
  config :myoption_key, :validate => :string, :default => 'myoption_value'
  public def register
  end
  public def run(queue)
  end
end
```

其中的大多数语句在过滤器和输出阶段是共有的。

config_name：用来定义该插件写在 Logstash 配置文件里的名字。

milestone：标记该插件的开发里程碑，一般为 1、2、3。如果是不再维护的，则标记为 0。

config：可以定义很多个，即该插件在 Logstash 配置文件中的可配置参数。Logstash 提供了验证方法，确保接收的数据是用户期望的数据类型。

register：定义 Logstash 在启动的时候运行的函数，一些需要常驻内存的数据可以在这一步完成。比如对象初始化、Filters/Ruby 插件中的 init 语句等。

2. 插件的关键方法

输入插件独有的是 run() 方法。在 run() 方法中，必须实现一个长期运行的程序（最简单的就是 loop 指令）。在每次收到数据并处理成 event 之后，一定要调用 queue << event 语句。这样，一个输入流程就算是完成了。

如果是过滤器插件，则对应修改为：

```
require 'logstash/filters/base'
class LogStash::Filters::MyPlugin < LogStash::Filters::Base
  public def filter(event)
  end
end
```

输出插件则是：

```
require 'logstash/outputs/base'
class LogStash::Outputs::MyPlugin < LogStash::Outputs::Base
  public def receive(event)
  end
end
```

另外，为了在终止进程的时候不遗失数据，建议都实现如下方法。只要实现

了该方法，Logstash 在关闭的时候就会自动调用。

```
public def teardown
end
```

17.3 ELK 在大数据运维系统中的应用

在海量日志系统的运维中，以下几个方面是必不可少的：

（1）分布式日志数据集中式查询和管理。

（2）系统监控，包含系统硬件和应用各个组件的监控。

（3）故障排查。

（4）安全信息和事件管理。

（5）报表功能。

ELK 功能模块如图 17-2 所示。它运行于分布式系统之上，通过搜集、过滤、传输、存储，对海量系统和组件日志进行集中管理和准实时搜索、分析；使用搜索、监控、事件消息和报表等简单易用的功能，帮助运维人员进行线上业务的准实时监控，业务异常时及时定位原因、排除故障，程序研发时跟踪分析 Bug，业务趋势分析，安全与合规审计，深度挖掘日志的大数据价值。同时 ElasticSearch 提供了多种 API（如 REST JAVA PYTHON 等）供用户扩展开发，以满足其不同需求。

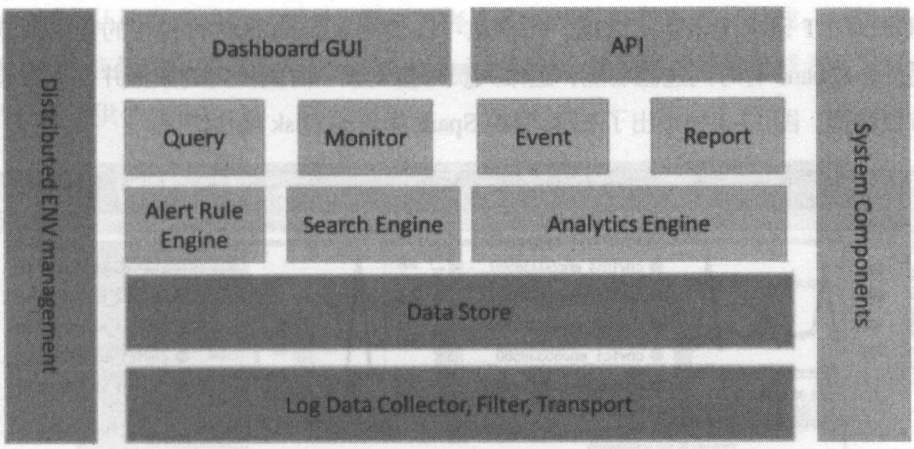


图 17-2 ELK 在运维系统中的应用图示

汇总 ELK 组件在大数据运维系统中主要解决的问题如下：

（1）日志查询，问题排查，上线检查。

（2）服务器监控，应用监控，错误报警，Bug 管理。

（3）性能分析，用户行为分析，安全漏洞分析，时间管理。

综上所述，ELK 组件在大数据运维系统中的应用是必不可少的，是一套易用的开源解决方案。

17.4 ELK 实战应用

17.4.1 ELK 监控 Spark 集群

通过 ELK 组件对 Spark 作业运行状态进行监控，搜集 Spark 环境下运行的日志，经过筛选、过滤并存储可用信息，从而完成对 Spark 作业运行和完成状态的监控，实时掌握集群状态，了解作业完成情况，并生成报表，方便运维人员监控和查看。

从数据源搜集到数据后，通过过滤形成固定的数据格式。目前支持过滤的类有 json、grep、grok、geoip 等，最后 Output 到数据库或者消息队列，比如 Redis、Kafka，或者直接传送给 ElasticSearch 存储。当数据被存储于 ElasticSearch 之后，用户可以使用 ElasticSearch 提供 DE API 来检索信息数据，如通过 REST API 执行 CURL GET 请求搜索指定数据。用户也可以使用 Kibana 进行可视化的数据浏览。另外，Kibana 有时间过滤功能，运维人员可查询某一时间段内的数据并生成报表，方便快捷。图 17-3 列举出了 ELK 监控 Spark 集群的 Disk 操作情况。

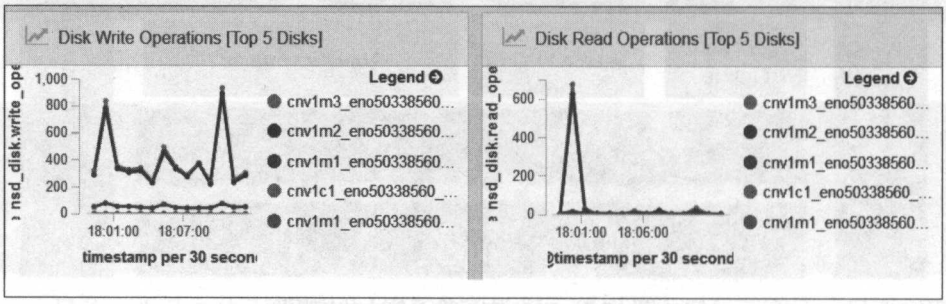


图 17-3　ELK 对 Spark Task 的监控

17.4.2 ELK 监控系统资源状态

可以通过 ELK 组件对系统资源状态进行监控。如图 17-4 所示是使用 ELK 组件为集群提供日志查询和系统资源监控的例子。通过各类日志搜集、分析、过滤、存储并通过 Kibana 展现给用户，供用户实时监控系统资源、节点状态、磁盘、CPU、MEM，以及错误、警告信息等。

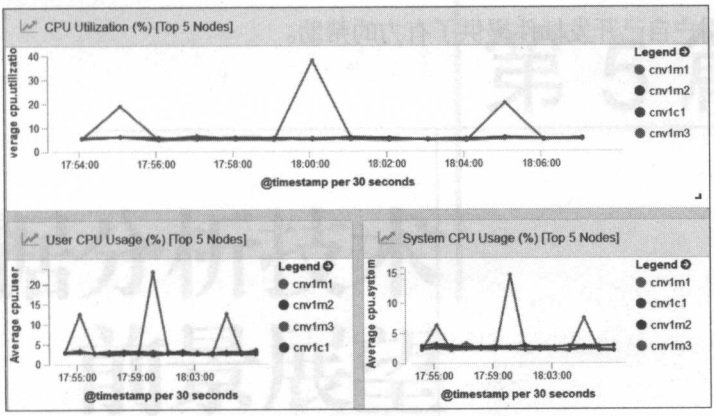


图 17-4 ELK 对系统状态进行监控

17.4.3 ELK 辅助日志管理和故障排查

可以通过 ELK 组件对系统进行日志管理和故障排查，如图 17-5 所示。用户可根据故障发生时间段集中查询相关日志，通过搜索、筛选、过滤等功能，快速定位问题，从而排查故障。另外，通过对各个应用组件的日志进行过滤，可快速列举出各个应用对应节点上的 Error 或 Warning 日志，从而对故障排查或者对发现产品 Bug 提供便捷途径。

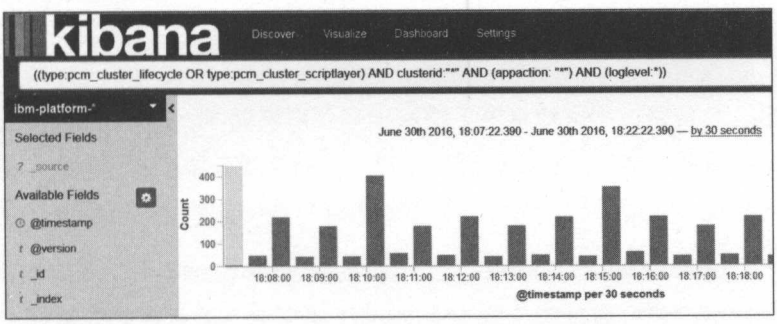


图 17-5 ELK 对日志的搜索和查询

17.5 小结

本章阐述了 ELK 的部署形式和优化方案，并展示了 ELK 应用的几个例子。事实上，ELK 已经应用于各种场合，包括 Hadoop 集群的监控、Spark 集群的监控等。在日常使用中，可能还会因为某种缺陷而无法达到用户的需求，这时候，我们可以根据 ELK 官方的方法来开发自己的插件。对于收集端 Beats，官方提供了接口和库，为用户自己开发插件提供了有力的帮助。

第5篇

数据分析技术前景展望

18 大数据处理的思考与展望

18.1 大数据时代的思考

让我们来回顾一下电影《永无止境》吧，这部电影很能说明大数据的威力。影片中，埃迪·莫拉本来是一个胸怀大志、充满抱负的作家，不断遭受创作方面的瓶颈所带来的痛苦。他无意中服用了一种叫作 NZT 的全新药物，之后，埃迪的生活发生了翻天覆地的变化。在药物的作用下，他的全部潜能被激发出来，每一个神经元的连接处都散发出耀眼的光芒，埃迪能够瞬间回忆起他所读过、看过或听过的任何事情，只用一天时间就能学会一门语言，多么复杂的方程式都难不倒他。这简直就是一种兼具人类和机器人的能力。

埃迪以旋风般的姿态迅猛地扫荡了华尔街，通过炒股，用微薄的资金连本带利地赚了数百万美元。埃迪是怎么炒股的呢？他能在短时间内掌握无数公司资料和背景，也就是将世界上已经存在的海量数据（包括公司财报、电视、几十年前的报纸、互联网、小道消息等）挖掘出来并串联起来，甚至通过对 Facebook、Twitter 的海量社交数据进行挖掘得到普通大众对某种股票的感情倾向，通过对海量信息的挖掘、分析，使一切内幕都不是内幕，使一切趋势尽在眼前。其实他的成功在很大程度上得益于“大数据”的帮助。

此影片提出的问题值得大数据界人士认真思考：人类通常只使用了大脑的

20%，如果剩余80%的大脑潜能被激发出来，世界会变得怎样？在企业、行业和国家管理中，通常只有效使用了不到20%的数据（甚至更少），如果剩余80%数据的价值被激发起来，世界会变得怎样？特别是随着海量数据的新摩尔定律，数据呈爆发式增长，如果数据都能得到有效应用，世界又会变得怎样呢？

简单地说，后大数据时代会是一个什么样的时代？从表面来看，大数据是一种不会“枯竭”的资源，借助这种资源，之前很多难以攻克的问题可能都不再是问题，之前只能靠运气、靠天意解决的问题，在大数据时代将变得可控。在大数据时代，我们的思维模式需要改变，我们的行为方式也需要改变。大数据到底会让我们的生活变得更加“简单”还是更加“复杂”？大数据会让我们的生活变得更加“有趣”还是更加“无聊”？比如现在的智能手机，到底是让我们的生活变得“精彩”还是“粗糙”了呢？

任何事情都有两面性，笔者认为，大数据也可能是把双刃剑，就看人类怎么利用。至于“大数据”到底会给人类带来多大的影响和挑战，留于后人评说吧。

18.2 大数据处理技术的发展趋势

目前大数据的处理方案众多，也有数以百计的工程师在日夜奋战，大数据已经从大公司独有的流行词和概念变成了驱动我们数字生活发展的动力。数据存储巨头EMC的CEO Pat Gelsinger透露，大数据处理目前的市场规模已达700亿美元，并且正以每年15%～20%的速度增长。几乎所有主要的大型科技公司都对大数据感兴趣，并对该领域的产品及服务进行了大量投入，其中包括IBM、Oracle、EMC、HP、Dell、SGI、日立、Yahoo!等。眼看这些大型企业动作频繁，越来越多的创业型公司也跃跃欲试。投资者正寻找大数据领域可靠的创业企业进行投资，创新孵化机构Accel就于2015年11月建立了1亿美元的“大数据”基金；同样，IA Ventures也于同年10月建立了同样的基金。

笔者认为，在未来几年，大数据处理的发展将有以下几个趋势。

1. Hadoop对MapReduce的依赖越来越小，并逐渐商业化

Hadoop平台只为MapReduce服务的时代从Hadoop 2.0版本开始正式结束了。新版本支持的产品和服务将会与Cloudera的Impala一样用一个SQL查询引擎或

者其他方法来补充甚至替代 MapReduce。HBase NoSQL 数据库就是一个很好的例子。大型网络公司，如 Facebook、eBay 等都已经使用 HBase 处理事务型的应用了。一部分较早的 Hadoop 项目将面临挑战。

2. 机器学习无处不在

机器学习越来越流行，如我们身边的小应用 Prismatic、Summly、Trifacta、CloudFlare、Twitter、Google、Facebook、Bidgely、Healthrageous、Predilytics、BloomReach、DataPop、Gravity……如今很难想象一家没有机器学习技术的科技公司可以生存。就连各大 IT 巨头，如 IBM、微软等，都在机器学习上下了很大赌注，认为它将成为公司一项重要的收入来源。

3. 在大数据处理中，最活跃的和最能满足客户需求的仍然是开源

开源软件将带来更多的市场机会：开源软件提供了解决问题的很多技术思路，并且完全免费，成本优势十分明显，且其蕴含的相关市场和商机巨大。

4. 大数据将由网络数据处理走向企业级应用

目前的大数据技术主要应用于 Google、Facebook、腾讯、百度、中国移动、中国联通这样的互联网或电信巨头。未来，大数据将不仅仅指的是网络数据处理，企业级应用对大数据处理的需求将会不断增加，包括数据流监测、数据分析等。

5. 大数据将推动基础架构向 Scale-out 发展

大数据需要分布式的方式，以实现更加有效的处理。传统的向上扩展型架构更适合处理结构化数据和关系型数据库，而大数据的处理则需要横向扩展型的基础架构。基础架构是大数据首先面临的挑战。如何让基础架构能够存取更多的数据？传统的基础架构能否满足用户的需求？从目前来看，虽然基础架构面临一些挑战，但挑战并不是很大。随着大数据行业的发展，数据将呈现爆发式增长，将对传统的架构形成巨大的冲击。随着数据量的逐渐增大，可以通过分布式的处理方式把复杂应用分散到分布式系统的各个节点上。传统的数据处理是由运算能力很强、CPU 主频非常高的一台机器来处理的，而不是像大数据这种多个节点、多个 CPU 核数来处理，这说明大数据时代的发展方向将从 Scale-up 转向 Scale-out。

6. NoSQL 市场将会重组，并与实时分析技术结合

NoSQL 在很多情况下又叫作云数据库。由于其处理数据的模式完全是分布于各种低成本的服务器和存储磁盘，因此它可以帮助网页和各种交互式应用快速处理海量数据。它为 Zynga、AOL、Cisco 及其他一些企业提供网页应用支持。正常的数据库需要将数据进行归类组织，类似于姓名和账号这些数据需要进行结构化和标签化。但是 NoSQL 数据库完全不关心这些，它能处理各种类型的文档。在处理海量数据的同时请求时，它也不会有任何问题。例如，如果有 1000 万人同时登录某个 Zynga 游戏，它会将这些数据分布于全世界的服务器并通过它们来进行数据处理，结果与 1 万人同时在线没什么两样。

现在有多种不同类型的 NoSQL 模式。商业化的模式如 CouchBase、10gen 的 MongoDB 及 Oracle 的 NoSQL；开源免费的模式如 CouchDB 和 Cassandra；还有亚马逊的 NoSQL 云服务。

目前 NoSQL 解决方案已经被大量用于具体的用户实例上，并且能够很好地运行，但这需要企业根据用户需求做大量的工作。在这种情况下，一些产品将被合并，一些将失去社区支持，一些将被其他供应商收购。

7. 新的技术将不断涌现

大数据处理技术不会因为采用了 Hadoop MapReduce、Spark 等技术而停止前行，相反，随着数据和场景的不断变化，新的技术还会不断涌现。今天我们动辄称呼“大数据”为一个时代，其实只是想要提醒尚未重视大数据发展趋势的企业或者组织，而事实上，大数据并非如此之大。也许我们不应该过分迷信大数据，大数据只是一个相对的概念，大数据之大是相对于传统数据及传统的数据处理技术而言的。但事实上，传统数据应用在刚刚兴起的时候也很“大”，传统的数据处理技术，如数据库、数据仓库、数据集市等，也在很大程度上解决了大规模数据的问题。今天，数据变得多元而复杂，传统技术、模型已经不再适用，总有一天，新的数据构建方法将发展成一个标准化的需求。到了那一天，今天所说的大数据将褪去“大”的外衣成为“单纯数据”，而未来也有比今天的大数据更大的数据。

18.3 小结

“大数据”已经在我们的身边。当你登录搜索网站的时候，一些搜索引擎公司会记录下你的搜索习惯，并根据这些信息自动向你推荐相关广告；当你在电子商务网站购物时，一些电商公司会依据你的购物喜好自动向你推荐有关产品；当你驾车上路时，电子导航系统会自动定位并为你指示方向……这些都得益于计算机对海量数据的存储、分析及处理。我们正生活在大数据时代，并享受着大数据为我们提供的便利。

Broadview
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

此书从广度上涵盖了目前几乎所有的大数据处理热门技术，实为大数据行业的入门人员和从业人员的一顿盛宴，推荐！

—— IBM 产品经理　张帅

目前大数据处理领域比较火爆的技术有 Hadoop、Spark、Storm 等，到底哪个技术较强？谁会替代谁？此书对这些技术进行了横向对比，分析了各个技术的应用场景，对大数据处理初学者和大数据行业应用人员有很好的指导作用。

—— YeahMobi 产品架构师　朱杰

大数据处理技术日新月异，从最初的 Hadoop 到目前广为流行的 Spark。与此同时，为了满足不同应用场景的需求，各大 IT 公司陆续推出不同的大数据处理方案，目前已有数十种大数据处理方案涌现。为此，我们只有不断学习新技术，才能跟上时代的步伐。本书对深入了解大数据处理技术有着很好的参考意义。

—— 华为大数据技术构架师　尹青

十年前，有人曾说过，“21 世纪什么最贵？”——“人才”，深以为然。十年后的今天，大数据的价值越来越明显，堪比石油和黄金。为了应对海量数据处理带来的巨大挑战，全球的计算机精英开发出众多有针对性的先进技术。本书全面介绍了目前大数据处理领域的热门技术的精髓和应用场景，实用性很强。

—— 中兴高达产品经理，技术总监　尹尚国

本书内容丰富、结构严谨、语言诙谐，为读者呈现了大数据处理领域各大热门技术的设计思想、应用场景和实例，可帮助大数据初学者快速上手，也可帮助大数据研发人员扩展知识，是大数据处理行业不可多得的书籍。

—— 知名 IT 领军人物　欧立奇



作者简介

何金池　IBM 高级软件工程师，硕士毕业于西安邮电大学，主要从事高性能计算和大数据领域研发工作，熟悉目前大数据处理领域的各项热门技术，擅长于大数据环境下的日志分析和处理。具有多年的一线软件研发测试的经验，曾经参与或负责了 PCM、HPC、ELK、Spark 等项目的研发和应用工作。

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

责任编辑：刘　皎
封面设计：吴海燕

欢迎投稿
邮箱：Ljiao@phei.com.cn
电话：010-88254395
新浪微博：@皎丫子

上架建议：大数据技术

ISBN 978-7-121-28723-7

9 787121 287237 >

定价：79.00元